Algorithme LLL polynomial et applications

Jean-René Reinhard

ECOLE POLYTECHNIQUE
PROMOTION X2000
REINHARD Jean-René

RAPPORT DE STAGE D'OPTION SCIENTIFIQUE

ALGORITHME LLL POLYNOMIAL ET APPLICATIONS

NON CONFIDENTIEL

Option :	Informatique
Champ de l'option :	Algorithmique
Directeur d'option :	Mr Jean-Marc STEYAERT
Directeur de stage :	Mr Guillaume HANROT
Dates du stage :	7 Avril - 4 Juillet
Adresse de l'organisme :	Projet SPACES, LORIA 615, rue du Jardin Botanique F-54602 Villers-les-Nancy Cedex

Abstract

In this report, we study the polynomial LLL algorithm and several of its applications. This algorithm was first published as such by A. K. Lenstra in 1985; it was then rediscovered by Paulus in 1996, but it was already more or less known as Popov normal form in linear algebra since 1969. A matrix M with coefficients in $K[X]$ being given, it amounts to find an invertible matrix U such that MU has minimal (in a precise sense) coefficients.

We implemented the polynomial LLL algorithm and tested it on two classes of examples, one with K a finite field of small characteristic (using machine integers), the other one with K a finite field of large characteristic (using GMP integers).

Then, we studied several applications of this algorithm : we implemented an algorithm for changing the monomial ordering of a Gröbner basis of an ideal of $K[X, Y]$; we studied an algorithm for the list-decoding of Reed-Solomon codes, derived from Boneh's algorithm for CRT list-decoding; finally, we studied an algorithm for bivariate polynomial factorization over a finite field, derived from van Hoeij's algorithm over $\mathbb{Z}$.

Résumé

Dans ce rapport, on étudie l'algorithme LLL polynomial et quelques-unes de ses applications. Cet algorithme, publié par Lenstra en 1985, redécouvert par Paulus en 1996 et existant plus ou moins sous le nom de forme normale de Popov en algèbre linéaire depuis 1969, consiste étant donné une matrice M à coefficients dans $K[X]$, à trouver une matrice U inversible telle que MU ait des coefficients en un certain sens minimaux.

Nous avons réalisé une implantation de l'algorithme LLL polynomial, que nous avons testée et évaluée sur des exemples avec K fini de petite caractéristique (entiers machine) et de grande caractéristique (entiers GMP).

Nous nous sommes ensuite intéressés à des applications de cet algorithme : un algorithme de changement d'ordre pour une base de Gröbner d'un idéal de $K[X, Y]$ que nous avons implanté; un algorithme de décodage en liste des codes de Reed-Solomon inspiré par l'algorithme de Boneh dans le cadre des codes CRT; enfin, un algorithme de factorisation de polynômes sur $K[X, Y]$, inspiré de l'algorithme de van Hoeij sur $\mathbb{Z}[X]$.

1 Introduction

L'algorithme LLL (Lenstra, Lenstra, Lovász), introduit en 1982 [12] pour résoudre le problème de la factorisation des polynômes à coefficients entiers en temps polynomial, est rapidement devenu un algorithme central du calcul formel et de la théorie algorithmique des nombres. L'objectif premier de cet algorithme est de calculer une base courte dans un réseau ; en quelques mots, cela revient à trouver de petites combinaisons linéaires à coefficients entiers de réels donnés.

En 1996, Paulus a proposé une version de cet algorithme s'appliquant à des polynômes en deux variables sur un corps fini. Cet algorithme a été assez peu utilisé depuis, à l'exception notable, en cryptologie, d'une méthode de calcul dans les jacobiennes de certaines courbes. En algèbre linéaire, cependant, on utilise depuis longtemps une forme de matrice, la forme de Popov [15], proche de la forme des matrices que Paulus fait intervenir. Au cours de recherche bibliographique on a constaté que A. K. Lenstra avait déjà utilisé l'algorithme LLL polynomial pour factoriser des polynômes bivariés [10].

L'objet du stage est d'implanter l'algorithme LLL polynomial, et d'étudier si certaines applications de l'algorithme LLL entier peuvent s'étendre dans ce cadre et fournir des résultats significatifs. Trois applications en particulier ont retenu notre attention.

Nous présentons d'abord une application qui trouve son origine dans les corps de fonctions. Les bases de Gröbner sont des familles génératrices particulières des idéaux de $K[X_1, \ldots, X_n]$. Grâce à elles, on peut rendre effectives les opérations $+, -, \times$ dans le quotient de l'anneau des polynômes par un idéal. Étant donné un idéal des polynômes multivariés muni d'un ordre monomial, le problème consiste à trouver, à partir d'une base de Gröbner de cet idéal pour cet ordre, une base de Gröbner pour un autre ordre monomial. Dans [1], A. Basiri et J-C. Faugère montrent comment, dans le cas bivarié, l'utilisation d'un algorithme LLL polynomial modifié permet de calculer une telle base à parir de la base de Gröbner réduite correspondant à l'ordre monomial initial. On se propose de réaliser une implantation en C de cet algorithme à partir de l'implantation de LLL polynomial.

Les deux applications suivantes reposent sur l'analogie générale entre les corps de nombres, l'arithmétique entière, et les corps de fonctions et l'arithmétique polynomiale. Comme le montre l'exemple du décodage en liste présenté ci-dessous, de nombreux procédés trouvant leur origine dans un cadre résolvent également des questions non-triviales dans l'autre cadre.

En 1999 M. Sudan et V. Guruswami ont proposé dans [9] un algorithme polynomial de décodage en liste des codes correcteurs de Reed-Solomon qui permet de récuperer un message à partir d'un code même lorsque le nombre d'erreurs dépasse la borne traditionnelle du code. Pour ce faire l'algorithme fournit la liste des messages possibles pour un nombre d'erreur inférieur ou égal à e. En 2002 D. Boneh a noté l'analogie entre l'interpolation des polynômes intervenant dans le décodage en liste des codes de Reed-Solomon et le théorème chinois, et a obtenu un résultat similaire pour le décodage en liste des codes CRT. Le cœur de cet algorithme est la recherche d'un petit élément dans un réseau entier, étape realisée par l'algorithme LLL entier ([3]). On se propose de transposer la méthode de D. Boneh dans le cadre polynomial, pour obtenir un nouvel algorithme de décodage en liste des codes de Reed-Solomon.

Le célèbre algorithme de Berlekamp-Zassenhaus permet de factoriser des polynômes à coefficients entiers. Au cours du calcul il soulève un problème combinatoire qu'il tente de résoudre en parcourant exhaustivement l'ensemble des parties d'un ensemble. En général la taille de cet ensemble est petite et l'algorithme se comporte bien. Cependant il existe des polynômes pour lesquels le coût est effectivement exponentiel en le degré du polynôme à factoriser. En 2002 van Hoeij a proposé dans [16] un algorithme qui permet de traiter le problème combinatoire rencontré ; cette méthode repose sur la réduction de réseau par l'algorithme LLL entier. On va tenter d'adapter cette méthode au cas de polynômes bivariés sur un corps fini K, que l'on peut encore voir comme des polynômes en Y à coefficients dans $K[X]$, X et Y étant les deux indéterminées.

2 Algorithme LLL polynomial

On présente dans cette section l'algorithme LLL polynomial. Cet algorithme opère sur des réseaux de $K[X]^n$ et permet d'obtenir des bases réduites de tels réseaux. On peut aussi le voir comme un procédé pour mettre une matrice M à coefficients dans $K[X]$ sous une forme dans laquelle la somme des normes des colonnes de la matrice est minimale. Pour ce faire il réalise des opérations inversibles sur les colonnes de la matrice de départ. On peut donc le voir comme processus permettant d'obtenir une matrice unimodulaire U tel que les colonnes de MU aient de petite norme, dans un sens que l'on précisera. On décrit plus en détail le fonctionnement de l'algorithme puis on présente l'implantation qu'on en a réalisée.

2.1 Fondements mathématiques

Soit K un corps et n un entier positif. Pour un polynôme $g \in K[X]$ on note $|g|$ le degré de g. La *norme* $|a|$ d'un vecteur $a = (a_1, \ldots, a_n) \in K[X]^n$ est définie comme $\max\{|a_i| : 1 \leq i \leq n\}$.

Soit $b_1, \ldots, b_n \in K[X]^n$ des vecteurs linéairement indépendants sur $K(X)$. Le *réseau* $L \subset K[X]^n$ de rang n engendré par $b_1, \ldots, b_n$ est l'ensemble des combinaisons linéaires des b_i à coefficients dans $K[X]$, soit

$$L = \sum_{j=1}^{n} K[X]b_j = \left\{ \sum_{j=1}^{n} r_j b_j : r_j \in K[X] \ (1 \leq j \leq n) \right\}.$$

Le *déterminant* $d(L) \in K[X]$ de L est défini comme le déterminant de la matrice $n \times n$ ayant les vecteurs $b_1, \ldots, b_n$ pour colonnes. Deux bases distinctes de L étant reliées par $\mathcal{B}_1 = \mathcal{B}_2 M$, avec $M \in GL_n(K[X])$, la valeur de $d(L)$ ne dépend pas de la base de L choisie, à une constante multiplicative près. Le *défaut d'orthogonalité* $OD(b_1, \ldots, b_n)$ d'une base $b_1, \ldots, b_n$ pour un réseau L est défini par

$$\left(\sum_{i=1}^{n} |b_i| \right) - |d(L)|.$$

On a $OD(b_1, \ldots, b_n) \geq 0$. En effet on peut écrire

$$d(L) = \sum_{\sigma \in S_n} \epsilon(\sigma) b_{\sigma(1),1} \cdots b_{\sigma(n),n},$$

or $\forall i, \left|b_{\sigma(i),i}\right| \leq |b_i|$ et

$$|d(L)| \leq \max_{\sigma \in S_n} \left\{ \sum_{i=1}^{n} \left|b_{\sigma(i),i}\right| \right\}.$$

On dit que la base $b_1, \ldots, b_n$ est *réduite* si $OD(b_1, \ldots, b_n) = 0$.

Une telle base correspond à une base de vecteurs courts au sens suivant : le déterminant étant indépendant de la base choisie à un élément non nul de K près, une base est réduite quand la somme des normes des vecteurs qui la composent est minimum.

Proposition 1 *Soit $b_1, \ldots, b_n$ une base pour un réseau L et notons $b_{i,j}$ la j-ème coordonnée de b_i. Si les coordonnées des vecteurs $b_1, \ldots, b_n$ peuvent être permutées de telle sorte qu'elles satisfassent*

1. *$|b_i| \leq |b_j|$ pour $1 \leq i < j \leq n$*
2. *$|b_{i,j}| < |b_{i,i}| \geq |b_{i,k}|$ pour $1 \leq j < i < k \leq n$*

alors la base $b_1, \ldots, b_n$ est réduite.

La deuxième condition peut être illustrée par la figure suivante, où la i-ème colonne de la matrice est b_i. Le j-ème élément dans la i-ème colonne donne la condition que satisfait $|b_{i,j}|$:

$$\begin{pmatrix}
= & |b_1| & < & |b_2| & < & |b_3| & \ldots & < & |b_n| \\
\leq & |b_1| & = & |b_2| & < & |b_3| & \ldots & < & |b_n| \\
\leq & |b_1| & \leq & |b_2| & = & |b_3| & \ldots & < & |b_n| \\
 & \vdots & & \vdots & & \vdots & & & \vdots \\
\leq & |b_1| & \leq & |b_2| & \leq & |b_3| & \ldots & = & |b_n|
\end{pmatrix}$$

On peut étendre cette théorie à des réseaux de rang inférieur à n, la norme du déterminant du réseau L, $|d(L)|$, étant alors remplacée par $\delta(L)$, le maximum des normes des déterminants des sous-matrices de tailles $m \times m$; le défaut d'orthogonalité s'écrit alors $(\sum_{i=1}^{m} |b_i|) - \delta(L)$. Enfin on peut aussi l'étendre au cas où on a non plus une base du réseau mais un système générateur.

2.2 Algorithme LLL de Paulus

L'algorithme LLL de Paulus calcule une base réduite d'un réseau à partir d'un système de vecteurs générateurs. Au cours de l'algorithme les coordonnées des vecteurs peuvent être permutées, mais on peut conserver la trace de ces permutations et revenir à l'ordre initial en fin de calcul. De plus l'algorithme ne fait intervenir que des calculs sur les coefficients, il ne nécessite pas d'arithmétique sur les polynômes.

L'opération élémentaire est la soustraction du produit par un monôme d'une colonne à une autre colonne ; elle correspond à l'élimination d'un «terme de tête». La tâche effectuée à

Algorithme 1 Algorithme LLL polynomial

$k \leftarrow 0$
while $k < l$ **do**
  Choisir $c \in \{b_1, \ldots, b_l\}$ tel que $|c| = \min\{|b_j| : k+1 \leq j \leq l\}$, soit i_c l'index correspondant, échanger(b_{k+1}, b_{i_c}).
  Résoudre $\sum_{i=1}^{k} a_{i,j,|a_i|} r_i = c_{j,|c|}$ pour $1 \leq j \leq k$ dans K
  $c' \leftarrow c - \sum_{i=1}^{k} r_i X^{|c|-|a_i|} \cdot a_i$
  if $|c'| = |c|$ **then**
    $a_{k+1} \leftarrow c$
    Permuter les coordonnées $(k+1, \ldots, n)$ de telle sorte que $|a_{k+1,k+1}| = |a_{k+1}|$
    $k \leftarrow k+1$
  else
    if $c' = 0$ **then**
      Eliminer b_{k+1}
      $l \leftarrow l-1$
    else
      $p \leftarrow \max\{0, \ldots, k : |a_l| \leq |c'|\}$
      for $j = k+1$ downto $p+2$ **do**
        $b_j \leftarrow a_{j-1}$
      end for
      $b_{p+1} \leftarrow c'$
      $k \leftarrow p$
    end if
  end if
end while

chaque étape est très simple : on choisit le vecteur de plus petite norme, on élimine par des opérations élémentaires ses composantes de plus haut degré situées au dessus de la diagonale dans la matrice. Si la norme du vecteur chute, on fait sortir de la base les vecteurs de norme supérieure à la norme du nouveau vecteur obtenu. Si le nouveau vecteur est nul, on l'élimine du système générateur du réseau. Si la norme reste égale à la norme du vecteur choisi au début de l'étape on permute les coordonnées afin que la composante de plus haut degré se retrouve sur la diagonale de la matrice.

Afin d'illustrer l'algorithme nous allons montrer comment il procède sur la matrice

$$\begin{pmatrix} 4x^2+3x+5 & 3x+6 & 6x^2+4x+2 \\ 4x^2+3x+4 & 3x+5 & 6x^2 \\ 6x^2+1 & x+3 & 2x^2+x \end{pmatrix}$$

à coefficients dans $\mathbb{F}_7[X]$. On note $(1),(2),(3)$ les colonnes de la matrice courante, $(a),(b),(c)$ les lignes de la matrice courante.

$$\begin{pmatrix} 4x^2+3x+5 & 3x+6 & 6x^2+4x+2 \\ 4x^2+3x+4 & 3x+5 & 6x^2 \\ 6x^2+1 & x+3 & 2x^2+x \end{pmatrix} \rightarrow \begin{pmatrix} 3x+6 & 4x^2+3x+5 & 6x^2+4x+2 \\ 3x+5 & 4x^2+3x+4 & 6x^2 \\ x+3 & 6x^2+1 & 2x^2+x \end{pmatrix}$$

inversion de (1) et (2), (1) entre dans la base, $k=1$.

$$\begin{pmatrix} 3x+6 & 4x^2+3x+5 & 6x^2+4x+2 \\ 3x+5 & 4x^2+3x+4 & 6x^2 \\ x+3 & 6x^2+1 & 2x^2+x \end{pmatrix} \rightarrow \begin{pmatrix} 3x+6 & 2x+5 & 6x^2+4x+2 \\ 3x+5 & x+4 & 6x^2 \\ x+3 & 3x+1 & 2x^2+x \end{pmatrix}$$

$(2)-6x(1)$, la norme de (2) a diminué, (2) n'entre donc pas dans la base.

$$\begin{pmatrix} 3x+6 & 2x+5 & 6x^2+4x+2 \\ 3x+5 & x+4 & 6x^2 \\ x+3 & 3x+1 & 2x^2+x \end{pmatrix} \rightarrow \begin{pmatrix} 3x+6 & 1 & 6x^2+4x+2 \\ 3x+5 & 6x+3 & 6x^2 \\ x+3 & 6 & 2x^2+x \end{pmatrix}$$

$(2)-3(1)$, la norme de (2) ne diminue pas, (2) entre donc dans la base, $k=2$.

$$\begin{pmatrix} 3x+6 & 1 & 6x^2+4x+2 \\ 3x+5 & 6x+3 & 6x^2 \\ x+3 & 6 & 2x^2+x \end{pmatrix} \rightarrow \begin{pmatrix} 3x+6 & 1 & 6x+2 \\ 3x+5 & 6x+3 & 4x \\ x+3 & 6 & 2x \end{pmatrix}$$

$(3)-2x(1)$, le degré de la deuxième composante de (3) est strictement inférieur à la norme initiale de (3), donc pas d'action de (2) sur (3) ($r_2=0$). la norme de (3) diminue, (3) n'entre donc pas dans la base, $k=2$.

$$\begin{pmatrix} 3x+6 & 1 & 6x+2 \\ 3x+5 & 6x+3 & 4x \\ x+3 & 6 & 2x \end{pmatrix} \rightarrow \begin{pmatrix} 3x+6 & 1 & 4 \\ 3x+5 & 6x+3 & 5x+4 \\ x+3 & 6 & 1 \end{pmatrix}$$

$(3) - 2(1)$ $(r_1 = 2)$

$$\begin{pmatrix} 3x+6 & 1 & 4 \\ 3x+5 & 6x+3 & 5x+4 \\ x+3 & 6 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 3x+6 & 1 & 2 \\ 3x+5 & 6x+3 & 5 \\ x+3 & 6 & 3 \end{pmatrix}$$

$(3) - 2(2)$ $(r_2 = 2)$, la norme de (3) a diminué elle n'entre pas dans la base.

$$\begin{pmatrix} 3x+6 & 1 & 2 \\ 3x+5 & 6x+3 & 5 \\ x+3 & 6 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 3x+6 & 1 \\ 5 & 3x+5 & 6x+3 \\ 3 & x+3 & 6 \end{pmatrix}$$

$(3) - 2(2)$ $(r_2 = 2)$, la norme de (3) a diminué elle n'entre pas dans la base. Comme la norme de (3) est devenu plus petite que la norme de vecteurs de la base, on fait sortir de la base les vecteurs de norme trop grande. $k = 0$

$$\begin{pmatrix} 2 & 3x+6 & 1 \\ 5 & 3x+5 & 6x+3 \\ 3 & x+3 & 6 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 6 & 1 \\ 5 & 6x+5 & 6x+3 \\ 3 & 3 & 6 \end{pmatrix}$$

(1) entre dans la base, $k = 1$. $(2) - 5x(1)$, le degré de (2) ne diminue pas, il entre dans la base, $k = 2$.

$$\begin{pmatrix} 2 & 6 & 1 \\ 5 & 6x+5 & 6x+3 \\ 3 & 3 & 6 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 6 & 2 \\ 5 & 6x+5 & 5 \\ 3 & 3 & 3 \end{pmatrix}$$

$r_1 = 0$, $r_2 = 1$, $(3) - (2)$, le degré de (3) a diminué, il n'entre pas dans la base

$$\begin{pmatrix} 2 & 6 & 2 \\ 5 & 6x+5 & 5 \\ 3 & 3 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 2 & 6 \\ 5 & 5 & 6x+5 \\ 3 & 3 & 3 \end{pmatrix}$$

la norme de (3) est devenue plus petite que certains vecteurs de la base, ces vecteurs sortent de la base, $k = 1$

$$\begin{pmatrix} 2 & 2 & 6 \\ 5 & 5 & 6x+5 \\ 3 & 3 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 0 & 6 \\ 5 & 0 & 6x+5 \\ 3 & 0 & 3 \end{pmatrix}$$

$(2) - (1)$.

$$\begin{pmatrix} 2 & 0 & 6 \\ 5 & 0 & 6x+5 \\ 3 & 0 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 6 & 0 \\ 5 & 6x+5 & 0 \\ 3 & 3 & 0 \end{pmatrix}$$

(2) s'annule, il sort du système générateur, $l = 2$, la matrice obtenue est réduite (la suite des opérations ne la modifie plus).

Remarque. Cet exemple n'a nécessité aucune permutation de ligne, car le coefficient déterminant la norme du vecteur était toujours situé sur la diagonale de la matrice après une opération. Il n'en est pas toujours ainsi.

2.3 Formalisme de Popov

On peut voir l'algorithme LLL de Paulus comme une manière d'ordonner une succession d'opérations élémentaires qui amènent une matrice quelconque dans la forme décrite par la proposition 1. On trouve dans [13] une forme de matrice très proche de celle décrite dans la proposition, la forme de Popov faible. Cette notion est bien connue en algèbre linéaire. L'algorithme de mise sous forme de Popov faible présenté dans cet article consiste en l'application de transformations élémentaires tant que celles-ci sont possibles. La terminaison de l'algorithme est prouvée par le fait que l'on peut associer à chaque colonne d'une matrice un élément d'un ensemble muni d'un ordre bien fondé et qu'une transformation élémentaire fait décroître cet élément pour un vecteur du système générateur. Cet algorithme de base permet d'obtenir des algorithmes donnant le rang, le déterminant, où encore une base du noyau de matrice à coefficients polynomiaux.

Afin de donner une preuve de la complexité de l'algorithme de Paulus, nous nous plaçons dans ce formalisme qui englobe le formalisme de Paulus. Nous reprenons dans ce qui suit les notations introduites dans [13], ainsi que la preuve de complexité qui s'y trouve. Les preuves des lemmes sont en général faciles.

Soit K un corps et $M = (m_{i,j}) \in K[X]^{n \times m}$.

Définition 1 *Pour $1 \leq j \leq m$ on appelle j-ème* indice pivot *de M et on note I_j^M l'indice défini comme suit :*

- *Si $m_{i,j} = 0$ pour $1 \leq i \leq n$ alors $I_j^M = 0$;*
- *Sinon I_j^M est l'unique indice vérifiant*
  1. *$\deg(m_{i,j}) < \deg(m_{I_j^M,j})$ pour $1 \leq i < I_j^M$;*
  2. *$\deg(m_{i,j}) \leq \deg(m_{I_j^M,j})$ pour $I_i^M < i \leq n$.*

Pour $I_j^M \neq 0$, l'élément $m_{I_j^M,j}$ est appelé j-ème *élément pivot* de M et est noté P_j^M. Le degré de P_j^M est appelé j-ème *degré pivot* de M et est noté D_j^M. Lorsque $I_j^M = 0$ on pose $D_j^M = -1$.

Définition 2 *On appelle ensemble porteur de M, et on note C^M, l'ensemble défini par $C^M = \{1 \leq j \leq m | I_j^M \neq 0\}$.*

Définition 3 *La matrice M est dite sous forme de Popov faible si ses indices pivots strictement positifs sont tous différents, c'est à dire*

$$k, l \in C^M, k \neq l \Rightarrow I_k^M \neq I_l^M.$$

L'intérêt de cette forme matricielle réside dans le théorème suivant.

Théorème 1 *Soit $b_1, \ldots, b_n \in K[X]^n$ une base d'un réseau L. Si la matrice dont les colonnes sont les vecteurs b_i est sous forme de Popov faible, alors la base est réduite.*

On veut pouvoir passer d'une matrice donnée à une matrice sous forme de Popov faible en appliquant des transformations sur ses colonnes.

Définition 4 *Si* $k \in C^M$,$l \neq k$ *et* $\deg(m_{I_k^M,l}) \geq D_k^M$, *il existe un unique scalaire* $c \in K$ *et* $e \in \mathbb{N}$ *tel que*

$$\deg(m_{I_k^M,l} - cx^e P_k^M) < \deg(m_{I_k^M,l}).$$

On appelle soustraire cx^e *fois la colonne* k *de la colonne* l *la* transformation simple *de la colonne* k *sur la colonne* l. *Si* $I_l^M = I_k^M$, *la transformation est dite de* première espèce, *sinon elle est dite de* deuxième espèce.

On donne à présent un algorithme de mise sous forme de Popov faible.

Algorithme 2 Mise sous forme de Popov faible

Entrée: $\mathcal{M} \in K[X]^{n \times m}$.
Sortie: $\mathcal{N}$ sous forme de Popov faible, obtenu en appliquant des transformations simples de première espèce à $\mathcal{M}$.

$\mathcal{A} \leftarrow \mathcal{M}$
while $\mathcal{A}$ n'est pas sous forme de Popov faible **do**
  appliquer une transformation simple de première espèce à $\mathcal{A}$
end while
$\mathcal{N} \leftarrow \mathcal{A}$
renvoyer $\mathcal{N}$

L'algorithme de mise sous forme de Popov faible est basé sur le lemme trivial suivant.

Lemme 1 *La matrice* M *n'est pas sous forme de Popov faible si et seulement si on peut appliquer une transformation élémentaire de première espèce à* M, *ce qui signifie que les indices pivots de* M *ne sont pas deux à deux distincts.*

Nous allons prouver dans la section suivante que cet algorithme est correct, c'est-à-dire qu'il termine. Nous obtiendrons simultanément une majoration sur le nombre d'opérations dans K réalisées au cours de l'exécution.

2.4 Complexité

Rappelons d'abord la correspondance entre les deux notions introduites. Clairement, si la matrice du réseau polynomial est de la forme explicitée dans la proposition 1, elle est sous forme de Popov faible, avec $I_j^M = j$ pour tout j. La réciproque n'est pas vraie. La forme définie dans la proposition est plus riche, car elle contient une information sur l'ordonnancement des vecteurs, par norme croissante, et permet de dégager les minima successifs du réseau. Néanmoins si la matrice du réseau est sous forme de Popov faible alors la base correspondante est réduite.

Comme nous l'avons déjà mentionné, l'algorithme LLL polynomial de Paulus est une suite de transformations simples de première espèce, et il s'arrête sur une matrice sous forme

de Popov faible ; c'est donc un déroulement possible de l'algorithme de mise sous forme de Popov faible. Une borne sur la complexité de l'algorithme 2 fournira également une borne sur la complexité de l'algorithme LLL polynomial.

Entamons donc la preuve de la terminaison de l'algorithme de mise sous forme de Popov faible, preuve qui nous donnera une complexité. On rappelle qu'elle est tirée de [13]. On utilise deux notations pour les matrices : M représente une matrice en toute généralité, $\mathcal{M}$ fait référence à une matrice intervenant dans l'algorithme 2.

Le lemme suivant décrit l'évolution des indices pivots au cours d'une transformation simple.

Lemme 2 *Soit N la matrice obtenue après l'application de la transformation simple de la colonne k sur la colonne l de M, transformation de première espèce. On a*
- *soit $D_l^N < D_l^M$;*
- *soit $D_l^N = D_l^M$ et $I_l^N > I_l^M$ (que l'on peut aussi écrire $n - I_l^N < n - I_l^M$).*

Il en découle le corollaire suivant, qui borne le degré des polynômes apparaissant au cours de l'algorithme.

Corollaire 1 *Si d est un majorant des degrés des polynômes apparaissant dans la matrice $\mathcal{M}$ alors les degrés des polynômes apparaissant dans les matrices $\mathcal{A}$ au cours de l'algorithme sont bornés par d.*

On décrit à présent l'ensemble des valeurs prises par le couple $(D_l^{\mathcal{A}}, I_l^{\mathcal{A}})$ pendant l'algorithme de mise sous forme de Popov faible.

Définition 5 *L'ensemble $I^M = \{I_j^M | j \in C^M\}$ des indices pivots non nuls de M est appelé ensemble des indices de M.*

Lemme 3 *Si N est la matrice obtenue après une transformation simple de M, alors $I^M \subseteq I^N$.*

Lemme 4 *Pour $1 \leq l \leq m$, les valeurs que peuvent prendre le couple $(D_l^{\mathcal{A}}, I_l^{\mathcal{A}})$ pendant l'algorithme sont dans l'ensemble $\{D_l^{\mathcal{N}}, D_l^{\mathcal{N}} + 1, \ldots, D_l^{\mathcal{M}}\} \times (I^{\mathcal{N}} \bigcup \{0\})$.*

Lemme 5 *Si les indices pivots de toutes les colonnes de M sont positifs et différents, alors les colonnes de M sont indépendantes sur $F(X)$.*

Corollaire 2 rang$(M) \geq \#I^M$.

Théorème 2 *L'algorithme de mise sous forme de Popov faible termine. Le coût de l'algorithme en opérations dans K est borné par $O(nmrd^2)$, où r est le rang de $\mathcal{M}$ et d une borne sur le degré de $\mathcal{M}$.*

Démonstration. D'après le lemme 4, le couple $(D_l^{\mathcal{A}}, I_l^{\mathcal{A}})$ peut prendre au maximum $(D_l^{\mathcal{M}} + 2)(\#I^{\mathcal{N}} + 1)$ valeurs. Comme rang($\mathcal{N}$)=rang($\mathcal{M}$), d'après le corollaire 2, $\#I^{\mathcal{N}} = r$. Par le lemme 2, on sait qu'une transformation élémentaire de première espèce fait décroître le couple $(D_l^{\mathcal{A}}, n - I_l^{\mathcal{A}})$ dans l'ordre lexicographique. Il s'en suit que le nombre de transformations simples appliquées au cours de l'algorithme est $O(mrd)$. Par le corollaire 1, le coût d'une transformation simple est $O(nd)$ opérations dans K. ■

Complexité de LLL polynomial En reprenant tout ce qui a été dit plus haut on conclut que l'algorithme LLL de Paulus est polynomial. Si l'on note

- n le nombre de vecteur dans le système générateur ;
- m la dimension de ces vecteurs ;
- r le rang de la matrice dont les colonnes sont les vecteurs de la base ;
- d un majorant du degré des polynômes dans la matrice ;

alors le coût de l'algorithme en nombre d'opérations dans K est $O(nmrd^2)$. Pour un réseau $L \subset K[X]^n$ de rang n le coût est $O(n^3d^2)$. La borne est plus fine que celle donnée par Paulus dans [14], qui donnait une borne en $O(nm^3d^2)$ en se basant sur un argument faisant intervenir la décroissance du défaut d'orthogonalité.

2.5 Implantation

Nous avons réalisé une implantation de l'algorithme LLL de Paulus en C. Nous nous sommes fixé comme objectif la rapidité d'exécution. On s'est efforcé en écrivant le code de respecter quelques règles simples, comme :

- Ne jamais allouer dynamiquement de la mémoire dans les fonctions effectuant des calculs. On demande plutôt en entrée de ces fonctions des pointeurs vers des zones de mémoires déjà allouées, dynamiquement ou statiquement, et on s'en sert comme des buffers pour stocker les données intermédiaires du calcul, lorsque c'est nécessaire. De plus lorsque c'est possible, on préfère les fonctions opérant en place.
- On exploite l'arithmétique des pointeurs pour éviter de recopier des opérandes.

Les polynômes sont représentés par une structure contenant le tableau de leurs coefficients et leur degré. Ceci impose, si l'on veut éviter d'avoir à effectuer des réallocations, d'allouer des tableaux de coefficients assez grands dès le départ. L'algorithme LLL possède une propriété tout à fait intéressante à cet égard : le degré des polynômes apparaissant dans une colonne de la matrice du réseau ne peut pas prendre une valeur plus grande que la norme initiale de cette colonne, considérée comme un vecteur.

Malgré cette limitation, cette représentation reste tout de même avantageuse, car aucune gestion de la mémoire n'est nécessaire au cours de l'algorithme. Si on avait adopté une représentation creuse des polynômes, sous forme de liste de coefficients, un temps très important serait passé dans la gestion des allocations/libérations de mémoire et dans le parcours de ces listes afin de les maintenir sous une forme standard (par degré décroissant par exemple). Si la taille des problèmes à traiter devient très importante, si la mémoire devient le facteur limitant de l'exécution, alors le passage à ce genre de représentation pourrait s'avérer nécessaire. On pourra économiser du temps sur la gestion de la mémoire en allouant une grande portion de mémoire dans le tas, et en créant une interface adéquate pour qu'elle ait le même comportement que le pile. On retrouve un tel procédé dans la librairie de PARI-GP (cf [2]), un logiciel de théorie des nombres.

On s'est efforcé d'écrire le code le plus général possible afin de pouvoir utiliser le même code pour différents corps de base ou différentes bibliothèques d'arithmétique (GMP, NTL, ...). Cela se traduit par la définition de macros dans un fichier séparé du reste du code, et

par leurs utilisations exclusive lorsqu'il s'agit de manipuler des éléments du corps, afin de pouvoir basculer facilement d'une représentation à une autre.

Les éléments du corps sont donc représentés par des *elt*, type de donnée défini dans le fichier de macros déjà évoqué. Les *elt* peuvent être de deux types : soit des types primitifs du langage, des entiers non signés par exemple, soit des pointeurs vers des structures de donnée. Dans les deux cas, on considère l'affectation du langage '=' comme une affectation «physique», c'est à dire que les zones mémoire pointées par les *elt* sont les mêmes. Pour obtenir une affectation dans le sens courant du langage, on utilise la macro *ASSIGN*. Pour tenir compte du fait que les *elt* peuvent être des pointeurs vers des structures de donnée, on définit également des macros *ALLOC* et *FREE* pour s'occuper de la gestion de la mémoire dans ce cas.

En ce qui concerne l'algorithme LLL polynomial, on ne résout pas explicitement le système triangulaire d'inconnues r_i présenté dans l'algorithme. En fait sa résolution est faite au fur et à mesure des calculs. On initialise c' avec c, puis si c'_1 est de degré la norme de c on effectue l'opération sur les colonnes qui supprime le terme le terme de plus haut degré de c'_1. Et on itère ce procédé avec $c'_2, \ldots, c'_k$. Ainsi, on économise la résolution du système linéaire, et on évite les opérations inutiles lorqu'un coefficient r_i est nul.

L'implantation a été testée pour des corps $K = \mathbb{F}_p$, avec p^2 inférieur à la longueur du mot machine. Les éléments du corps sont représentés par des entiers non signés C et leur caractéristique. Que ce soit en suivant pas à pas la progression du programme sur des exemples simples, ou en l'exécutant sur des exemples générés aléatoirement le programme donne des résultats satisfaisants, en conformité avec les complexités issues de l'analyse théorique. Nous donnons ici des valeurs de temps d'exécution obtenus pour diverses valeurs de n et d. On effectue ces tests sur un PIII 500 tournant sous Linux.

Variation du temps d'exécution en fonction de N=n=m. Pour une valeur de d fixée on donne les temps d'exécution du programme de réduction LLL sur des matrices carrés aléatoires de polynômes de tailles N dans TAB 1.

TAB. 1 – temps d'exécution pour $d = 20$, N variable (t en s)

N	t
20	0.01
50	0.07
100	0.66
150	2.38
200	5.70

Ce qui donne la courbe en Fig-1.

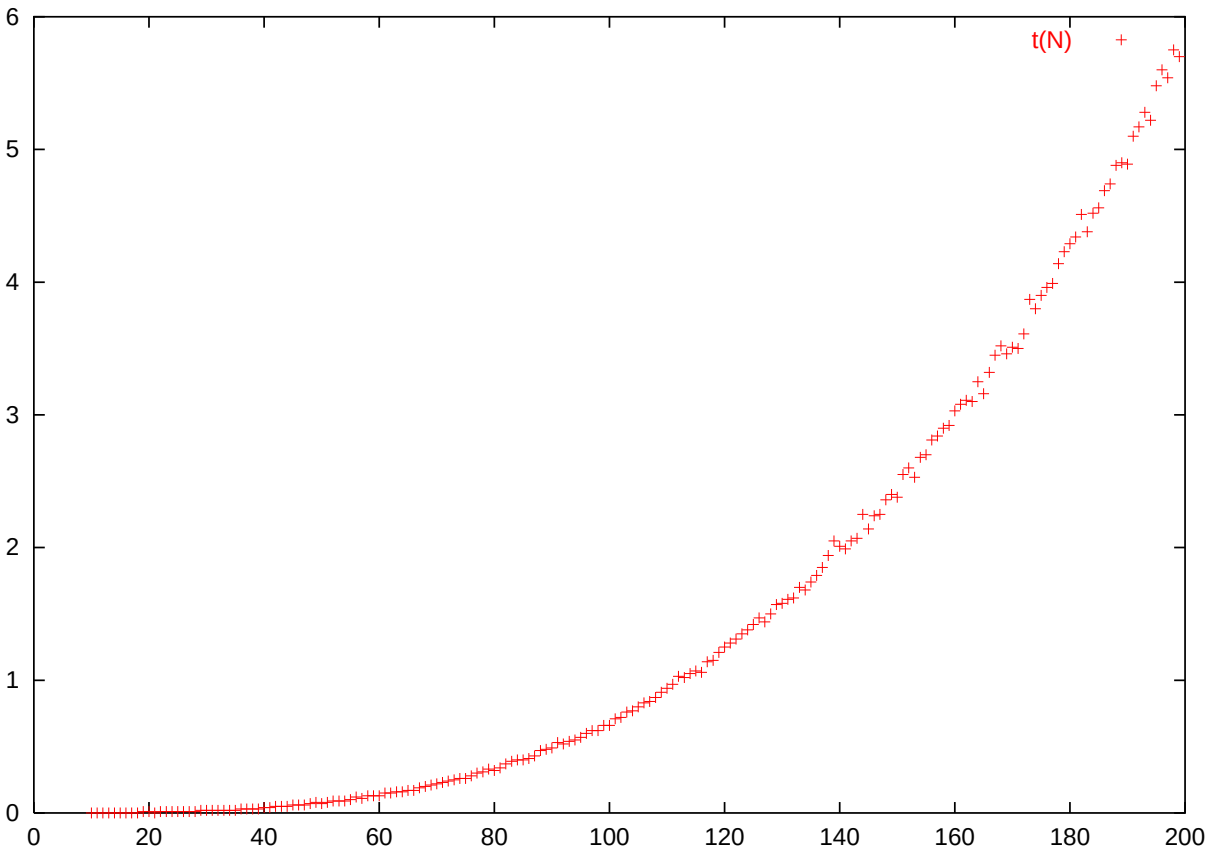

FIG. 1 – $t(N)$ pour N variant de 10 à 200, $d = 20$ ($t(N)$ en s)

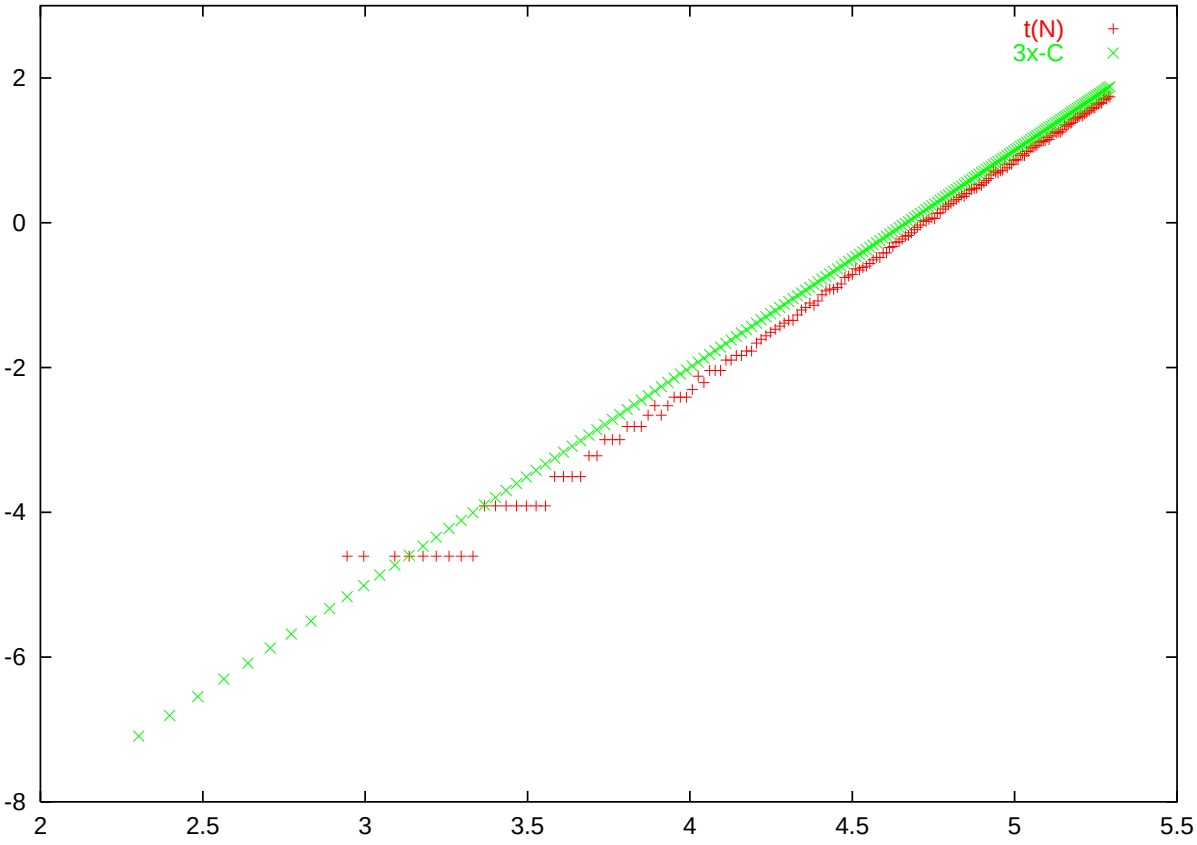

FIG. 2 – $t(N)$ pour N variant de 10 à 200, en coordonnées logarithmiques, $d = 20$

Pour dégager la nature cubique de la courbe, on la trace en coordonnées logarithmiques, ainsi qu'une fonction affine de pente 3 (voir Fig-2)

Les résultats expérimentaux sont en accord avec une dépendance cubique en N.

Variation du temps d'exécution en fonction de d. Pour une valeur de N fixée on donne les temps d'exécution du programme de réduction LLL sur des matrices carrés aléatoires de polynômes de degré au plus d en Tab 2.

TAB. 2 – temps d'exécution pour $N = 50$, d variable, pour des matrices aléatoires (t en s)

d	t
20	0.07
50	0.17
100	0.34
150	0.56
200	0.70

Ce qui donne la figure Fig-3.

Dans ce cas la dépendance semble linéaire en d. Ceci est lié au fait qu'une matrice aléatoire est «presque réduite», c'est à dire $D_l^{\mathcal{M}} - D_l^{\mathcal{N}} = 0(1)$ d'où une complexité en $O(nmrd)$. Une matrice du type

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ p_{1,1}(x) & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p_{N-1,1}(x) & p_{N-1,2}(x) & \cdots & 1 & 0 \\ p_{N,1}(x) & p_{N,2}(x) & \cdots & p_{N,N-1}(x) & p_{N,N}(x) \end{pmatrix},$$

où $p_{i,j}(x)$ est un polynôme aléatoire de degré d, demande plus de travail, qu'une matrice aléatoire générale, puisqu'à la fin de l'algorithme les termes diagonaux devront avoir des degrés de l'ordre de d/N. On donne donc le temps d'exécution de l'algorithme sur de telles matrices en Tab 3, ce qui donne la figure Fig-4.

On trace également la courbe en coordonnées logarithmiques sur la figure Fig-5.

Là encore on n'observe pas de dépendance quadratique, mais une dépendance en $0(d^{1,5})$.

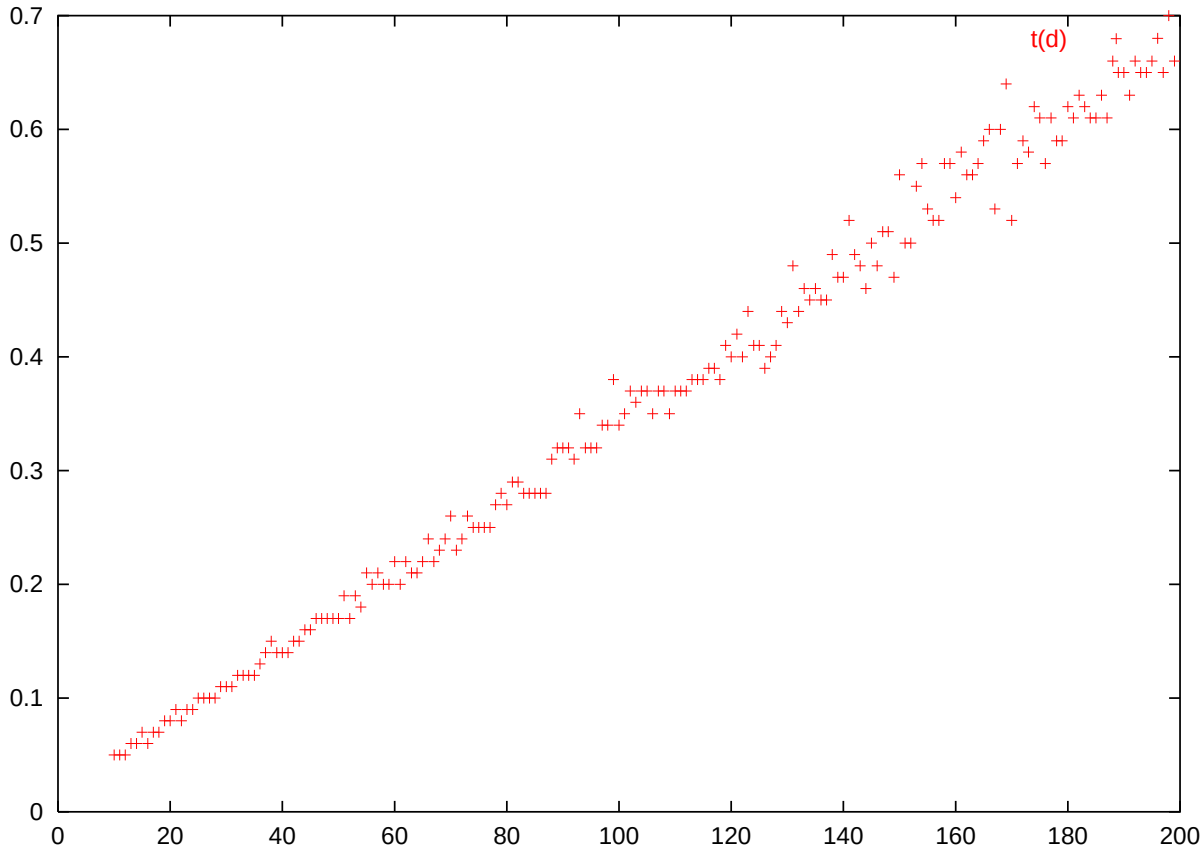

FIG. 3 – $t(d)$ pour d variant de 10 à 200, $N = 50$($t(d)$ en s)

TAB. 3 – temps d'exécution pour $N = 50$, d variable, pour des matrices aléatoires triangulaires(t en s)

d	t
20	0.49
50	1.53
100	4.12
150	7.68
200	12, 40

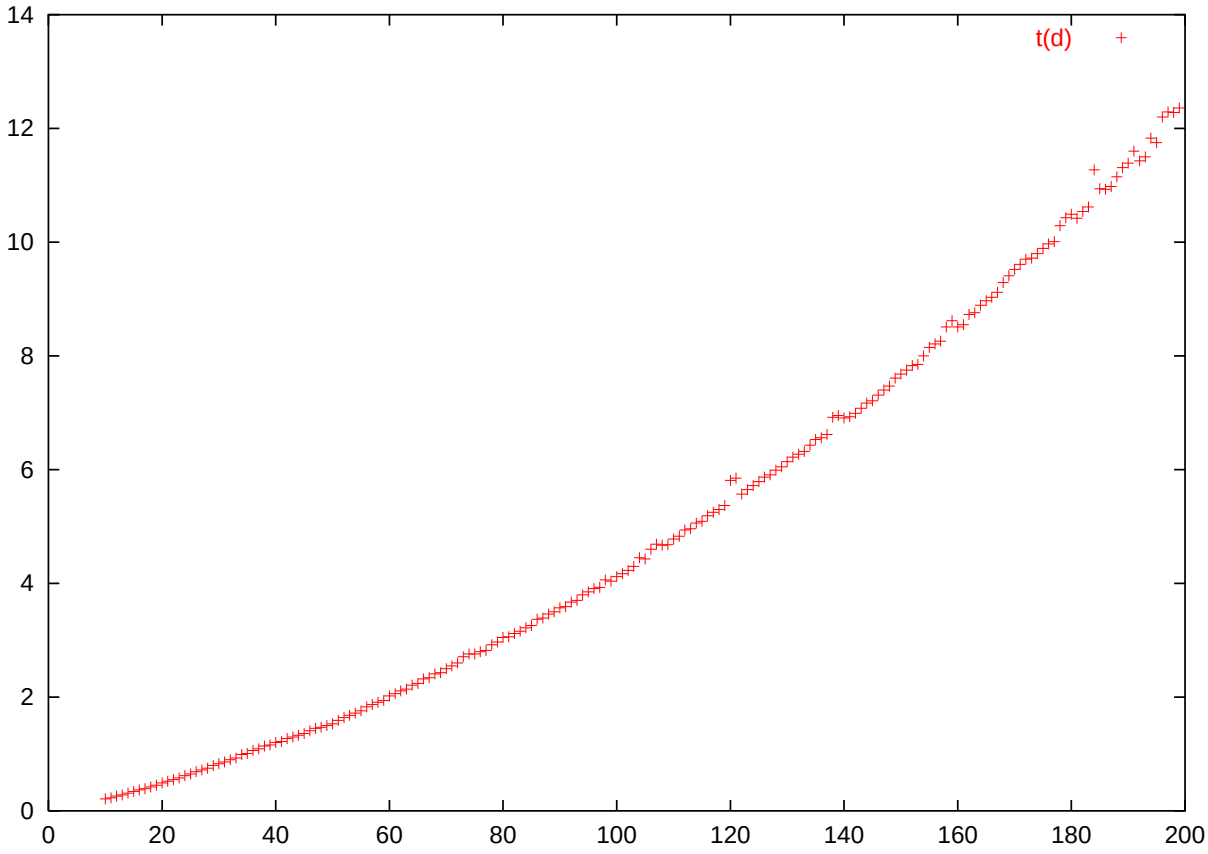

FIG. 4 – $t(d)$ pour d variant de 10 à 200, $N = 50$ ($t(d)$ en s

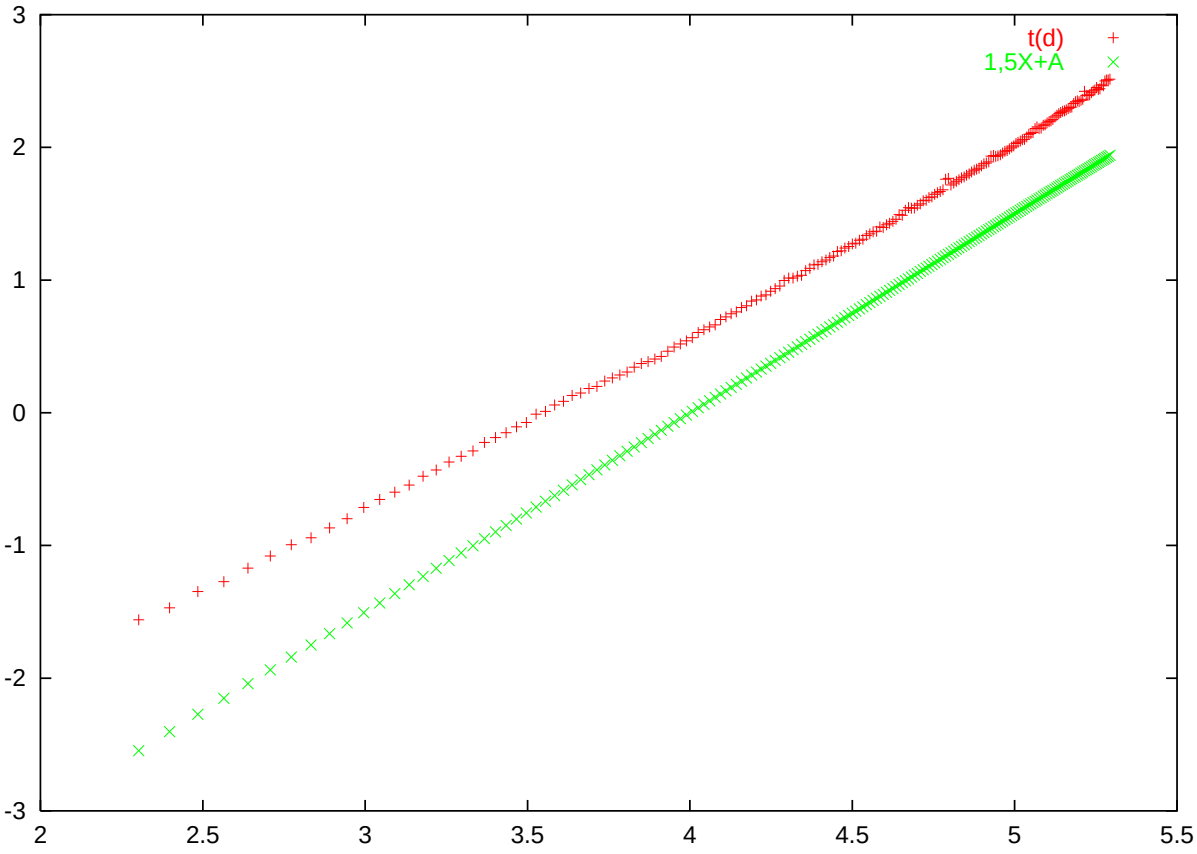

FIG. 5 – $t(d)$ pour d variant de 10 à 200, $N = 50$

Utilisation de gmp La bibliothèque GMP (GNU Multiple Precision)[1], développée principalement par T. Grandlund, est la bibliothèque multiprécision de référence à l'heure actuelle. Elle s'organise en plusieurs couches :

- une couche "de base", manipulant directement des pointeurs sur des mots (`mpn`), dont un certain nombre de fonctions sont écrites directement en assembleur hautement optimisé pour la plupart des processeurs courants ;
- une couche "entière", manipulant une structure permettant de représenter les entiers multiprécision (`mpz`) ; cette structure contient naturellement, outre le pointeur, la taille de l'entier correspondant, son signe, ainsi, pour des raisons pratiques, que la taille réellement allouée pour le pointeur. Son implantation utilise naturellement la couche `mpn` ; elle intègre certaines des techniques arithmétiques les plus performantes (Karatsuba, Toom-Cook, FFT, division rapide, etc.).

On dispose ainsi naturellement de toutes les fonctions permettant de disposer d'une arithmétique modulaire simple (notre modèle d'implantation permet d'utiliser GMP au prix d'un fichier `.h` d'une vingtaine de lignes) et extrêmement efficace.

On se contente de donner les temps d'exécution pour quelques valeurs des paramètres pour donner une idée de l'ordre de grandeur des problèmes que l'on peut traiter. On donne ces temps pour des caractéristiques p de l'ordre de 2^{32}, 2^{64}, 2^{128} et 2^{256}.

	ordre de grandeur de p			
N	2^{32}	2^{64}	2^{128}	2^{256}
20	0.29	0.46	0.64	1.14
40	3.45	2.94	3.97	6.90
60	5.59	8.5	11.50	19.95
80	12.22	18.51	25.01	43.39
100	22.07	33.23	46.36	81.25

TAB. 4 – temps d'exécution en fonction de N, $d = 20$ pour différentes tailles de coefficients (en s)

[1] `http://www.swox.se/gmp`

	ordre de grandeur de p			
d	2^{32}	2^{64}	2^{128}	2^{256}
20	0.05	0.08	0.12	0.21
40	0.15	0.24	0.36	0.62
60	0.30	0.5	0.71	1.23
80	0.51	0.82	1.18	2.04
100	0.76	1.23	1.76	3.03

TAB. 5 – temps d'exécution en fonction de d, $N = 10$ pour différentes tailles de coefficients (en s)

3 Applications possibles

Dans cette partie, nous présentons trois problèmes que l'algorithme vu dans la section précédente va nous permettre d'aborder, en exposant leur motivation et en donnant les idées qui permettent d'adapter l'algorithme LLL polynomial à ces problèmes.

3.1 Changement d'ordre monomial dans les bases de Gröbner

3.1.1 Introduction

Introduites par Buchberger dans sa thèse en 1965, les bases de Gröbner permettent d'effectuer les opérations élémentaires dans les quotients d'anneaux de polynômes multivariés, ou encore de résoudre les systèmes d'équations polynomiales.

Dans le cas d'une seule variable, tous les idéaux sont engendrés par un seul polynôme et l'algorithme d'Euclide permet donc de mettre les éléments d'un quotient $K[X]/I$ sous une forme normale. Mais pour pouvoir étendre l'algorithme de division avec reste, il faut

- ordonner les monômes suivant un ordre total, «stable» par multiplication et qui étende l'ordre partiel de la divisibilité. Un tel ordre total est dit monomial.
- disposer d'une base de l'idéal qui possède une certaine propriété, une base de Gröbner pour cet ordre monomial.

Définition 6 *Soit $<$ un ordre sur l'ensemble des monômes de $K[X_1, \ldots, X_n]$. On dit que l'ordre $<$ est monomial s'il satisfait les propriétés suivantes :*

- *$<$ est une relation d'ordre totale ;*
- *pour tous monômes n, m, m' de $K[X_1, \ldots, X_n]$, $m < m'$ implique $nm < nm'$;*
- *pour tous monômes n, m de $K[X_1, \ldots, X_n]$, si $n \neq 1$ alors $m < nm$.*

Exemples Parmi les ordres monomiaux utilisés dans la littérature et dans la pratique on trouve les ordres suivants ; ils jouissent de propriétés sur lesquelles nous ne nous étendrons pas ici. Soient donc $\mathbf{X}^\alpha$ et $\mathbf{X}^\beta$ deux monômes avec $\alpha = (\alpha_1, \ldots, \alpha_n)$ et $\beta = (\beta_1, \ldots, \beta_n)$.

- *Ordre lexicographique(lex).* On dit que $\mathbf{X}^\beta < \mathbf{X}^\alpha$ si $\alpha_i > \beta_i$ pour le plus petit indice i tel que $\alpha_i \neq \beta_i$. C'est l'ordre du dictionnaire sur les «mots» α et β.

– *Ordre du degré raffiné par l'ordre lexicographique inverse (drl).* On dit que $\mathbf{X}^\beta < \mathbf{X}^\alpha$ pour cet ordre si $\sum \beta_i < \sum \alpha_i$, ou, en cas d'égalité, si $\alpha_i < \beta_i$ pour le plus grand indice i tel que $\alpha_i \neq \beta_i$.

Définition 7 *Soit $<$ un ordre monomial sur $K[X_1, \ldots, X_n]$ et $P \in K[X_1, \ldots, X_n]$. on appelle terme de tête de P et on note $lt(P)$ le terme de P le plus grand selon $<$. On appelle coefficient dominant et on note $lc(P)$ son coefficient, on appelle monôme dominant et on note $lm(P)$ le monôme correspondant.*

Définition 8 *Soit $<$ un ordre monomial sur $K[X_1, \ldots, X_n]$ et I un idéal de $K[X_1, \ldots, X_n]$. On appelle* base de Gröbner *de I pour l'ordre monomial $<$ toute famille de polynômes $G_1, \ldots, G_l$ telle que le terme de tête de tout élément de I(le plus grand de ses monômes selon $<$) est divisible par le terme de tête d'un élément de cette famille.*

Pour $F \in K[X_1, \ldots, X_n]$, l'algorithme de division appliqué à $F, G_1, \ldots, G_n$ consiste à effectuer les opérations suivantes : tant qu'un terme de F est divisible par le terme de tête de l'un des G_i, éliminer ce terme en soustrayant à F le multiple de G_i qui convient. En gardant trace de ces opérations on obtient des quotients Q_i et un reste R tel que $F = Q_1G_1 + \ldots + Q_lG_l + R$ et aucun des termes de R n'est divisible par le terme de tête de l'un des G_i.

Algorithme 3 Division avec reste dans $K[X_1, \ldots, X_n]$

Entrée: $F_1, \ldots, F_s, G \in K[X_1, \ldots, X_n]$ et un ordre monomial $<$.
Sortie: Des polynômes $Q_1, \ldots, Q_s, R$ tels que $G = Q_1F_1 + \cdots + Q_sF_s + R$.

```
Poser Q_1 = ··· = Q_s = R = 0, L_i = lt(F_i)
while G ≠ 0 do
  L_G = lt(G)
  if il existe i tel que L_G est divisible par L_i then
    Q_i = Q_i + L_G/L_i
    G = G − (L_G/L_i)F_i
  else
    G = G − L_G
    R = R + L_G
  end if
end while
```

Pour tout ordre monomial, tout idéal admet une base de Gröbner finie. Quitte à rajouter des conditions de minimalité, cette base devient unique. Si aucun terme de polynôme de cette base n'est divisible par le terme de tête d'un autre polynôme, on dit que la base est réduite. On a le résultat d'unicité suivant :

Proposition 2 *Soit F dans $K[X_1, \ldots, X_n]$. Il existe un unique R combinaison K-linéaire de monômes non divisibles par les termes de tête des G_i tel que $F = Q + R$, avec $Q \in I$. L'algorithme de division appliqué à F et $G_1, \ldots, G_n$ calcule R, quels que soient les choix effectués au cours de l'algorithme.*

Le problème difficile est l'obtention d'une base de Gröbner d'un idéal. L'algorithme de Buchberger ([4]) et les algorithmes F4 ([7]) et F5 ([8]) de Faugère permettent de passer d'un système générateur de l'idéal à une base de Gröbner pour un ordre monomial, mais leur complexité est exponentielle en le degré des polynômes du système générateur. Cependant, en pratique, ils se comportent mieux vis-à-vis de certains ordres monomiaux. Il peut donc être intéressant dans certains cas de calculer une base de Gröbner pour un ordre convenable, puis de passer d'un ordre monomial à un autre. D'où l'intérêt d'utiliser un algorithme polynomial pour passer d'une base de Gröbner d'un idéal pour un ordre $<_1$ à une base de Gröbner de cet idéal pour un ordre $<_2$. FGLM, un algorithme dû à Faugère, Gianni, Lazard et Mora, réalise cette opération [6].

3.1.2 Algorithme de changement d'ordre monomial

Basiri et Faugère ([1]) ont également proposé un autre algorithme basé sur l'algorithme LLL polynomial. On le donne ici dans le cas de deux variables. L'algorithme repose sur l'injection canonique :

$$\begin{array}{rcl} K[X]^{D_Y} & \rightarrow & K[X,Y] \\ (v_1,\ldots,v_n) & \rightarrow & \sum_{j=1}^{D_Y} v_j Y^{j-1} \end{array}$$

et sur le résultat ci-dessous tiré de [1]. On commence par donner quelques définitions

Définition 9 *Soit $(G = (g_1,\ldots,g_m), <)$ une base de Gröbner réduite de I. On note r_i le degré du terme de tête de g_i par rapport à Y. On peut permuter les indices $(1,\ldots,m)$ de telle sorte que $r_i \leq r_{i+1}$. Pour tout entier $D_Y \geq \deg_Y(G)$ on définit*

$$B_{D_Y}(G) = \{Y^{j_i} g_i | 1 \leq i \leq m,\ 0 \leq j_i \leq r_{i+1} - r_i - 1\}$$

où $r_{m+1} = D_Y$. On note $M_{D_Y}(G)$ le $K[X]$ sous-module de $K[X,Y]$ engendré par $B_{D_Y}(G)$. $M_{D_Y}(G)$ est le D_Y-ème $K[X]$ module associé à I par rapport à $<$.

Le théorème fondamental s'écrit alors très simplement :

Théorème 3 *Le D_Y-ème $K[X]$ module associé à I par rapport à $<$ est égal à l'ensemble des polynômes de l'idéal I de degré strictement plus petit que D_Y.*

Si l'on sait borner le degré en Y de la base de Gröbner pour le nouvel ordre monomial par D_Y, on peut voir tout élément de l'idéal comme le produit par une puissance de Y d'un élément du $K[X]$ module de $K[X,Y]$, donc comme un réseau de $K[X]^{D_Y}$. Pour obtenir une base de Gröbner pour le nouvel ordre, il suffit de réduire la base B_{D_Y} obtenu à partir de la base de Gröbner pour l'ancien ordre. Mais on n'élimine plus les termes suivant leur degré en X mais suivant l'ordre monomial $<_2$.

ModuleBasis est la fonction qui calcule la base du réseau à réduire à partir de la base de Gröbner réduite pour l'ordre initial.

En notant D_X le degré en X de la base finale cet algorithme effectue $O(D_Y^3 \cdot D_X^2)$ opérations dans K.

Algorithme 4 Algorithme LLL polynomial modifié pour le changement d'ordre monomial

Entrée: $(G_{old} = (g_1, \ldots, g_m), <_{old})$ une base de Gröbner réduite pour I,$<_{new}$ et D_Y un entier positif assez grand.

Sortie: $G_{new} = (a_1, \ldots, a_l)$ une base de Gröbner pour I par rapport à l'ordre $<_{new}$.

$(b_1, \ldots, b_l) \leftarrow \text{ModuleBasis}(G_{old}, <_{old}, D_Y)$
$k \leftarrow 0$
while $k < l$ **do**
  choisir $i_0 \in \{k+1, \ldots, l\}$ tel que $b_{i_0} = \min_{<_{new}}\{b_i : k+1 \leq i \leq l\}$, **échanger**$(b_{k+1}, b_{i_0})$
  choisir $j \in \{1, \ldots, D_Y\}$ tel que $lt_{new}(b_{k+1}) = lt_{new}(b_{k+1,j})$
  if $j \leq k$ **then**
    $c \leftarrow \frac{lc_{new}(b_{k+1})}{lc_{new}(b_{a_j})} X^{\deg(b_{k+1,j}) - \deg(a_{j,j})} a_j$
  else
    $c \leftarrow b_{k+1}$
  end if
  if $lt_{new}(c) = lt_{new}(b_{k+1})$ **then**
    $a_{k+1} \leftarrow c$
    permuter $(k+1, \ldots, n)$ de telle sorte que $lt_{new}(a_{k+1,k+1}) = lt_{new}(a_{k+1})$
    $k \leftarrow k+1$
  else
    $p \leftarrow \max\{0 \leq s \leq k : a_s \leq_{new} c\}$
    for $i = k+1$ downto $p+2$ **do**
      $b_i \leftarrow a_{i-1}$
    end for
    $b_{p+1} \leftarrow c$
    $k \leftarrow p$
  end if
end while

3.1.3 Implantation

Nous sommes partis de l'implantation de l'algorithme LLL-polynomial général, et l'avons adapté afin que la sélection des vecteurs et des composantes au cours de l'algorithme ne se fasse plus suivant leur degré en X, mais suivant un ordre monomial, paramètre de la fonction. Nous avons adopté la convention de C, qui représente un ordre comme une fonction prenant en argument les deux objets à comparer et renvoyant -1,0 ou 1, suivant que le premier argument est plus petit, égal ou supérieur au deuxième.

Comme précedemment, les polynômes de $K[X]$ sont représentés de manière dense, par le tableau de leurs coefficients et leur degré. Les vecteurs de polynômes représentent à présent des polynômes bivariés, la i-ème composante du vecteur étant le coefficient à valeur dans $K[X]$ de Y^i. Habituellement mal adaptée aux polynômes multivariées, cette représentation nous est utile, car comme nous l'avons déjà évoqué, elle permet d'effectuer les opérations au cours de l'algorithme LLL sans recourir à une utilisation abusive de la mémoire. On adjoint de plus à la représentation d'un vecteur les degrés en X et en Y de son terme de tête. Etant donné que l'indice de la composante dans le vecteur code la valeur du degré en Y du polynôme, on garde une trace des échanges de lignes réalisés, afin d'accéder rapidement à l'indice $r[i]$ de ligne qu'occupait initialement la ligne i.

On a ensuite écrit la fonction *ModuleBasis* qui à partir de la base de Gröbner réduite initiale calcule la base du module $M_{D_Y}(G)$ qui sera réduit par la version modifiée de LLL.

D'autre part, nous disposions d'exemples fournis sur la page web de J-C. Faugère (`http://www-calfor.lip6.fr/~jcf/Papers/`) qui donnent les bases de Gröbner pour quelques idéaux pour l'ordre DRL et l'ordre lexicographique. Ces bases étant données sous forme réduite, on a implanté une version élémentaire d'algorithme de réduction d'une base de Gröbner, s'appuyant sur l'algorithme de division présenté plus haut. En effet, la sortie de l'algorithme LLL est bien une base de Gröbner, mais comme elle n'est pas réduite, on peut difficilement vérifier la correction de la sortie sans transformation supplémentaire. Nous n'avons pas chercher à optimiser cette fonction, son but n'étant que de fournir une vérification de la sortie.

3.1.4 Résultat

Étant donné la forme que prend la complexité de l'algorithme, on a intérêt à minimiser D_Y. On procède donc à un échange entre le rôle des variables X et Y dans le cas où $D_X > D_Y$, en tenant bien sur compte de ce changement de rôle au niveau des ordres monomiaux. Par exemple, si l'ordre monomial était l'ordre lexicographique, on adopte l'ordre lexicographique inverse ($Y > X$).

Nous avons dans un premier temps fait tourner notre programme sur les exemples cités plus haut. Les temps obtenus peuvent être comparés au temps de calculs de l'implantation MAPLE réalisé par Faugère et Basiri (voir Tab. 6). L'utilisation de C permet de gagner approximativement un facteur 100 sur le temps d'éxecution de l'algorithme. Le gain n'est pas surprenant dans la mesure où le C est un langage de plus bas niveau que MAPLE et qu'on ne fait pas d'allocations mémoire pendant l'exécution de l'algorithme pour ne pas pénaliser le temps de calcul, ce qui n'est pas le cas dans MAPLE.

TAB. 6 – Comparaison des temps d'exécution des implantations C et MAPLE de l'algorithme

DRL -> LEX	LLL(C) en sec	LLL(MAPLE) en sec	D_X	D_Y
benchmark D1	$0,01$	$0,15$	48	2
cyclic5	$0,01$	$0,25$	15	8
Uteshev Bikker	$0,03$	$1,5$	36	8
Fabrice24	$0,03$	$1,6$	40	8
dessin2	$0,04$	$1,8$	42	8
benchmark i1	$0,09$	$6,4$	66	11
cyclic6	$0,05$	$3,15$	48	12
katsura7	$0,40$	$40,2$	128	15
katsura8	$5,29$	$356,8$	241	22
random 50	$0,03$	$2,6$	50	9
random 100	$0,17$	$17,5$	100	13
random 150	$0,77$	$62,6$	150	16
random 200	$2,15$	$157,2$	200	19
random 300	$11,78$	$766,7$	300	24

D'autre part, nous avons tenté de suivre l'évolution des composantes au cours de l'algorithme, ainsi que les calculs effectués. Pour un passage de l'ordre DRL à l'ordre lexicographique, après une première phase «d'ordonnancement», on passe par une phase où l'on rencontre des paliers assez longs (cf Fig. 6) qui correspondent à une succession d'opérations élémentaires entre deux vecteurs. Ceci s'explique assez facilement par un exemple.

Exemple. Plaçons nous dans $\mathbb{F}_7 = \mathbb{Z}/7\mathbb{Z}$, l'ordre monomial considéré est l'ordre lexicographique et considérons la matrice suivante :

$$\begin{array}{c} \begin{array}{cc} (1) & (2) \end{array} \\ \begin{pmatrix} X & X^4+X^3+X^2 \\ X & X \end{pmatrix} \end{array} \begin{array}{l} \\ (Y=0) \\ (Y=1) \end{array}$$

Rien ne garantit qu'elle corresponde à une base de Gröbner d'un quelconque idéal de $\mathbb{F}_7[X][Y]$. Néanmoins sa forme peut être rencontrée au cours de l'algorithme et en lui appliquant une suite d'opérations élémentaires on peut se rendre compte de ce qui se passe.

$$\begin{pmatrix} X & X^4+X^3+X^2 \\ X & X \end{pmatrix} \quad \begin{array}{c} (2)-X^3*(1) \\ \longrightarrow \end{array} \quad \begin{pmatrix} X & X^3+X^2 \\ X & 6X^4+X \end{pmatrix} \quad \begin{array}{c} (2)-X^2*(1) \\ \longrightarrow \end{array}$$

$$\begin{pmatrix} X & X^2 \\ X & 6X^4+6X^3+X \end{pmatrix} \quad \begin{array}{c} (2)-X*(1) \\ \longrightarrow \end{array} \quad \begin{pmatrix} X & 0 \\ X & 6X^4+6X^3+6X^2+X \end{pmatrix}$$

On voit que tant que le degré en X de la première composante du vecteur (2) est supérieur ou égal à 1, on continue à faire une opération sur le vecteur (2) à partir du vecteur (1), car

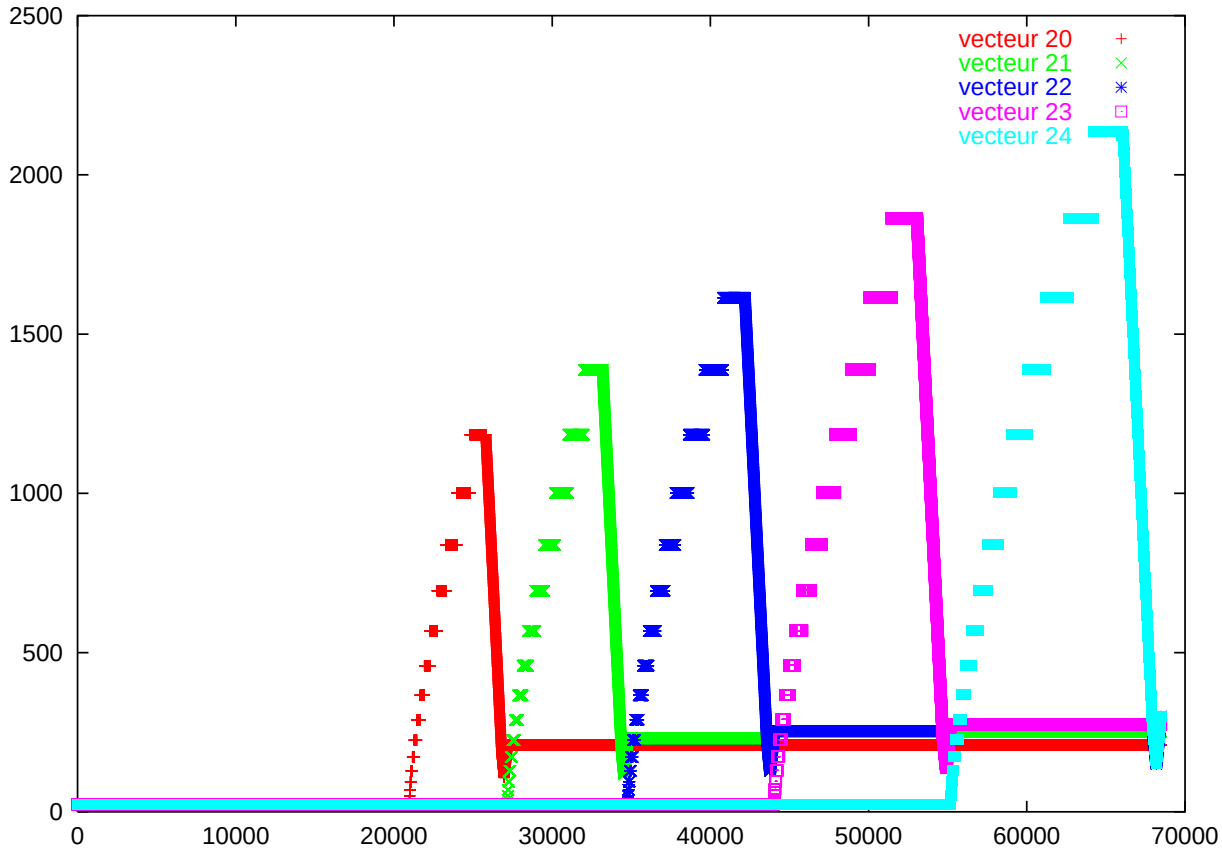

FIG. 6 – Évolution des normes des vecteurs au cours de LLL polynomial modifié

le terme de tête ne change pas de place, ni ne devient plus petit que les termes de tête des polynômes bivariés de la base.

Remarques. On constate que le degré en X des polynômes n'est plus borné comme dans le cas de l'algorithme LLL-polynomial classique et qu'il peut augmenter fortement. Sur les exemples considérés, cette divergence a été traitée en allouant un espace mémoire très important. Bien entendu cela n'est pas satisfaisant dans le cas général. Pour traiter de gros problèmes, des bases de Gröbner intervenant dans des applications cryptographiques par exemple, une gestion intelligente de la mémoire est aussi nécessaire que la rapidité de calcul. On peut tout de même remarquer que l'explosion du degré en X ne concerne qu'un seul vecteur, les normes des autres restant bornés par une borne D_X qui découle des propriétés des bases de Gröbner, et des ordres monomiaux de départ et d'arrivée. On peut donc imaginer d'appliquer un traitement spécial à la gestion de ce vecteur courant. Mais comme on l'a vu par ailleurs, l'algorithme LLL n'est qu'une manière d'ordonner une suite d'opérations élémentaires conduisant à la réduction de la base du réseau fourni en entrée de l'algorithme. On pourrait imaginer de trouver un nouvel ordonnancement de cette série d'opérations qui assure une borne sur les normes en X des vecteurs.

Une deuxième remarque est que, dans le cas où on utilise le même pivot plusieurs fois d'affilée, la suite des opérations élémentaires réalisées peut se voir comme les étapes d'une division de polynômes classique. On peut donc tenter d'accélérer le processus en remplaçant cette division quadratique par un algorithme de division rapide. Un tel algorithme repose sur la connaissance d'algorithmes de multiplication rapide, c'est-à-dire sous-quadratique. En effet dans l'algorithme de division classique, on élimine successivement les termes de têtes du dividende en lui soustrayant le produit par un monôme du diviseur. Ce faisant on oublie que l'on sait multiplier en temps sous-quadratique le diviseur par un polynôme.

3.1.5 Arithmétique rapide sur les polynômes

Les polynômes sont le cadre naturel des opérations rapides telles que la multipication ou la division par exemple. Les algorithmes appliqués aux polynômes s'étendent aux entiers, en les considérant comme des polynômes dont l'indéterminée est la base, mais il faut tenir compte des retenues.

Dans ce qui va suivre nous allons présenté un algorithme de multiplication rapide que nous avons implanté, puis un algorithme de division rapide qui utilise cet algorithme. Enfin nous discuterons de son application à notre exemple.

Multiplication. On appelle *longueur* d'un polynôme la taille du tableau de ses coefficients.

Il s'agit de multiplier deux polynomes de $F[X]$, $p = \sum_{i=0}^{n-1} p_i X^i$ et $q = \sum_{i=0}^{m-1} q_i X^i$, de longueur respective n et m. Le résultat de la multiplication est un polynôme de longueur $n + m - 1$. La méthode naïve consiste à se servir de la définition du polynôme produit comme étant le polynome dont les coefficients sont les $r_k = \sum_{i=0}^{k} p_i q_{k-i}$. Cette méthode conduit à effectuer nm multiplications. Si l'on multiplie deux polynômes de même longueur on obtient une multiplication en $0(n^2)$. Il existe des méthodes qui permettent d'obtenir des complexités asymptotiques sous quadratiques. L'algorithme de multiplication de Karatsuba

permet d'obtenir une multiplication de deux polynomes de longueur n en $O(n^{1.59})$ multiplications élémentaires, celui de Schönhage et Strassen permet d'obtenir une multiplication en $O(n\log(n)\log(\log(n)))$ multiplications élémentaires. Ces complexités sont asymptotiques. Les constantes se cachant sous la notation O font qu'en pratique on fait appel à tous ces algorithmes, le choix de l'algorithme employé étant déterminé par la taille des polynômes en argument. Ainsi pour des tailles faibles on utilise la multiplication naïve, puis pour des tailles intermédiaires la multiplication de Karatsuba, et pour des tailles élevées la multiplication de Schönage et Strassen

On présente ici de manière plus détaillée l'algorithme de multiplication de Karatsuba, ou plutôt devrait-on dire la famille d'algorithme de multiplication de Karatsuba.

Généralités On s'appuie ici sur [5] L'idée fondamentale de l'algorithme de Karatsuba est de remplacer une multiplication de polynômes de taille n et m par 3 multiplications de polynômes de taille moitié. On procède ainsi pour découper les polynômes $p = \sum_{i=0}^{n-1} p_i X^i$ et $q = \sum_{i=0}^{m-1} q_i X^i$:

On écrit $p = p_a + tp_b$ et $q = q_a + tq_b$, pour t que l'on déterminera plus tard. On a alors $pq = p_a q_a + t((p_a + p_b)(q_a + q_b) - p_a q_a - p_b q_b) + t^2 p_b q_b$.

Version plus haut degrés - plus bas degrés Dans cette version il faut que $n = m$. p_a(resp. q_a) est le polynôme constitué des termes de plus bas degré de p(resp. q), p_b(resp. q_b) est le polynôme constitué des termes de plus haut degré de p(resp. q) et $t = X^{\lceil \frac{n}{2} \rceil}$. En notant $K(n,m)$ le coût d'une multiplication de polynômes de longueur respectives n et m en nombre de multiplications élémentaires, on obtient la relation de récurrence

$$K(n,n) = 2K(\lceil \frac{n}{2} \rceil) + K(\lfloor \frac{n}{2} \rfloor)$$

Version pair - impair Dans cette version la séparation du polynome en deux polynômes de taille moitié ne s'opère plus selon les termes de haut degré et de bas degré, mais en degrés pair et impair. On écrit cette fois ci $F_n[X] = O_{\lceil \frac{n}{2} \rceil}[X] + XO_{\lfloor \frac{n}{2} \rfloor}[X]$, avec $O_k[X] = \{P(X^2)/P \in F_k[X]\}$. On remarque que la valeur que prend t, c'est-à-dire X, ne dépend plus de la longueur des polynômes. On peut alors utiliser cette méthode directement sur des polynômes de longueurs différentes. Une variante légerement modifiée de la version pair-impair l'améliore légèrement. Elle consiste à réaliser la multiplication de Xp par q si le polynôme p est de longueur impair. En fait la multiplication par X est traitée par un décalage qu'on fait intervenir dans l'algorithme, et la sortie de l'algorithme dans ce cas est pq.

On obtient alors la relation de récurrence :

$$K(n,m) = K(\lceil \frac{n}{2} \rceil, \lceil \frac{m}{2} \rceil) + K(\lfloor \frac{n}{2} \rfloor, \lceil \frac{m}{2} \rceil) + K(\lceil \frac{n}{2} \rceil, \lfloor \frac{m}{2} \rfloor)$$

Implantation - comportement pratique Dans [5], on trouve une preuve qui montre qu'en théorie, la version pair-impair modifiée de l'algrithme de Karatsuba est la version la plus rapide. Nous avons implanté ces deux algorithmes. Une utilisation astucieuse de

la mémoire permet de réaliser ces opérations en utilisant un buffer auxiliaire de taille $2\max(n,m)+O(1)$.

Si on utilise une représentation d'élément du corps F sous forme d'entier C, contrairement à ce qu'annonce le résultat théorique, la version la plus rapide est la version plus haut degré - plus bas degré. En effet, dans la version pair - impair, on a effectivement moins d'opérations sur les coefficients des polynômes, mais on effectue plus d'opérations en manipulant les coefficients des tableaux, afin d'extraire les parties paires et impaires des polynômes passés en argument.

Division rapide Maintenant que nous disposons d'une multiplication rapide, nous exposons un algorithme de division rapide, l'algorithme de division récursive. On reprend largement la présentation qui en est faite dans le rapport de recherche [18].

Soit à diviser un polynôme A de longueur $n+m$ par un polynôme B de longueur n, donnant un quotient de longueur m et un reste de longueur au plus $n-1$. Ces longueurs se déduisent facilement de la définition du quotient et du reste comme les polynômes vérifiant $A=QB+R$, $\deg(R)<\deg(B)$ et $\text{longueur}(P)=\deg(P)+1$ pour tout polynôme P. Si $m>n$, on commence par diviser les $2n$ termes de plus haut degré du dividende par le diviseur, ce qui donne les n termes de plus haut degré du quotient, et un nouveau dividende de taille inférieure ; on continue jusqu'à ce que le dividende soit de longueur $\leq 2n$. Si au contraire $m<n$, on divise les $2m$ mots de plus haut degré du dividende par les m termes de poids fort du diviseur, ce qui donne une approximation Q' du quotient, que l'on corrige éventuellement si la longueur du reste est égale à n.

L'opération de base est la division d'un polynôme de taille $2n$ par un polynôme de taille n. Nous présentons dans ce qui va suivre un algorithme de division récursive qui utilise la multiplication de Karatsuba.

L'algorithme de division naïve s'écrit pour $n=2$ de la façon suivante, où la fonction divrem retourne quotient et reste de la division euclidienne d'un dividende de longueur 2 par un diviseur de longueur 1.

Algorithme 5 division naïve pour $n=2$

Entrée: $A=a_3X^3+a_2X^2+a_1X+a_0$, $B=b_1X+b_0$ avec $b_1\neq 0$.
Sortie: quotient Q et reste R de la division de A par B.

$(q_1,r_1)\leftarrow$ **divrem**$(a_3+a_2,b1)$
$A\leftarrow r_1X^2+a_1X+a_0-q_1b_0X$
$(q_0,r_0)\leftarrow$ **divrem**$(a'_2X+a'_1,b1)$ où $A=a'_2X^2+a'_1X+a'_0$
$A\leftarrow r_0X+a'_0-q_0b_0$
renvoyer $Q=q_1X+q_0$, $R=A$

Cet algorithme fonctionne récursivement, en remplaçant X par X^n. La division d'un polynôme de $4n$ mots par $2n$ mots nécessite alors deux divisions récursives de $2n$ mots par n mots, et deux multiplications $n\times n$ à savoir q_1b_0 et q_0b_0.

Si $D(n)$ désigne la complexité de la division $2n$ par n, et $M(n)$ celle de la multiplication $n\times n$, on obtient la récurrence suivante :

$$D(2n)\sim 2D(n)+2M(n),$$

dont la solution est $D(n) \sim \frac{1}{2^{\alpha-1}-1} M(n)$ si $M(n) = \Theta(n^\alpha)$ avec $\alpha > 1$. Pour $\alpha = \log_2 3$ (Karatsuba), cela donne $D(n) \sim 2M(n)$.

Cet algorithme s'étend facilement au cas où n n'est pas une puissance de 2 : on remplace alors les deux divisions de taille $n/2$ par une division de taille $\lceil \frac{n}{2} \rceil$ et une division de taille $\lfloor \frac{n}{2} \rfloor$.

Nous avons implanté cet algorithme de division rapide d'un polynôme de taille $2n$ par un polynome de taille n, et nous nous en sommes servis pour écrire l'algorithme de division d'un polynôme de taille $n+m$ par un polynôme de taille n. En utilisant des éléments de corps représenté par des entiers C, on retrouve l'ordre de grandeur théorique $D(n) \sim 2M(n)$.

Application à LLL dans le cas du changement d'ordre monomial Nous avons incorporé un branchement dans l'algorithme LLL afin que l'opération d'une colonne sur une autre n'élimine non plus un terme de tête en opérant uniquement sur les termes de têtes, mais plusieurs simultanément. Pour cela, si on doit faire opérer la colonne j sur la colonne i et si on peut déterminer qu'on répète cette opération un grand nombre de fois, alors on n'effectue pas l'opération élémentaire, mais on calcule le quotient et le reste obtenu par la division euclidienne de $b[j][i]$ par $b[j][j]$, et on fait l'opération $b[i] - qb[j]$. Ainsi on est censé opérer plus rapidement. Hélas ce n'est pas le cas. En effet, en pratique $m \gg n$, puisque comme nous l'avons vu la norme des vecteurs diverge fortement. Dans ce cas la manière de découper le dividende pour pouvoir appliquer une succession de division rapide fait perdre l'avantage de disposer d'une multiplication rapide. Si on note $m = (k-1)n$, la division d'un polynôme de longueur kn par un polynôme de longueur n demande $k-1$ divisions de polynômes de longueur $2n$ par n et on obtient $D(kn, n) \sim 2(k-1)M(n)$, avec n petit. Là encore, le problème de la croissance des normes des vecteurs de polynômes du réseau est gênante.

3.2 Décodage en liste des codes de Reed-Solomon

3.2.1 Introduction

Un code correcteur d'erreur C, de longueur N, de taille de mot initial K et de distance D sur un alphabet q-aire Σ (un $[N, K, D]_q$ code) est une fonction de Σ^K, l'espace des messages, dans Σ^N, l'espace des mots du code. Cette fonction est telle que la distance de Hamming (nombre de coordonnées différentes) entre deux éléments de l'image de cette fonction est supérieure ou égale à D.

Un code de Reed-Solomon est un code correcteur dont les paramètres sont de la forme $(n, k+1, n-k)$; Σ étant un corps fini F. Un message peut être vu comme un polynôme de degré inférieur ou égal à k. Le code C associe à ce polynôme son évaluation en n points distincts de F.

Le problème du décodage pour un code $[N, K, D]_q$ consiste à trouver un mot du code dont la distance de Hamming à un mot donné, *reçu*, est plus petite qu'une borne e. Pour un code de Reed-Solomon, tant que $e < \frac{N-K+1}{2}$, on peut décoder en temps polynomial. On voit aussi que dès que $e \geq \frac{N-K+1}{2}$, il peut y avoir plus d'un mot du code à distance e du mot reçu, et l'algorithme de décodage peut ne pas renvoyer une solution correcte s'il ne renvoie qu'une seule solution.

Ceci motive le problème du décodage en liste : celui-ci consiste à trouver la liste de *tous* les mots du code à une distance $\leq e$ du mot reçu. Le décodage en liste peut permettre de récupérer un nombre d'erreurs allant au-delà de la borne traditionnelle du code (i.e. la quantité $\lfloor \frac{D-1}{2} \rfloor$).

Dans [3], D. Boneh montre comment on peut utiliser LLL-entier pour résoudre le problème du décodage en liste CRT, analogue pour les entiers du décodage en liste d'un code correcteur de Reed-Solomon. Par analogie, en utilisant LLL-polynomial on peut obtenir un algorithme polynomial de décodage en liste des codes de RS pour des valeurs de $e < n - \sqrt{nk}$.

3.2.2 Application de LLL polynomial

Le problème de décodage en liste s'écrit : étant donnés $\langle x_1, \ldots, x_n \rangle$,$\langle y_1, \ldots, y_n \rangle$, k, t : trouver les polynômes Q de $F[X]$ de degré plus petit que k tels que $Q(x_i) = (y_i)$ pour au moins t valeurs de i.

L'idée de l'algorithme est de construire un polynôme bivarié $w(X, Y)$ tel que pour tout polynôme p appartenant à la solution du problème du décodage en liste, $w(X, p(X)) = 0$. Pour ce faire on commence par chercher un polynôme $W \in \{Q \ / \ \forall p, Q(X, p(X)) = 0 \mod \Pi\}$, où Π est un polynôme de $F[X]$. Plus précisément, on cherche W dans un sous-module de dimension finie de cet ensemble engendré par une famille de polynômes déterminés. Puis on utilise une propriété qui dit en substance que si le degré d'un polynôme bivarié Q est petit, alors $Q(X, p(X)) = 0 \mod \Pi \Rightarrow Q(X, p(X)) = 0$. Or l'algorithme LLL polynomial permet précisément de déterminer un polynôme de petit degré dans un module. On peut donc calculer un polynôme w de petit degré. Ceci étant fait, dans une deuxième phase, on résout l'équation $w(X, p(X)) = 0$ dans $F[X]$, et on retient les racines solutions du problème.

On note $P(X)$ le polynôme $\Pi_{i=1}^{n}(X - x_i)$ et $R(X)$ un polynôme vérifiant $\forall i, R(x_i) = y_i$.

Définition 10 *Soit m un polynôme de $F[X]$. On appelle* amplitude *de m la quantité définie par*

$$\mathrm{amp}(m) = \gcd(P(X), m - R).$$

Remarque. le degré de $\mathrm{amp}(m)$ est le nombre de i tels que $m(x_i) = y_i$.

On considère maintenant les deux familles de vecteurs suivantes :

$$\begin{array}{rcll} g_i(X, Y) & = & P(X)^{a-i} \cdot (Y - R(X))^i & \textit{pour } i = 0, \ldots, a-1, \\ h_i(X, Y) & = & (Y - R(X))^a \cdot Y^i & \textit{pour } i = 0, \ldots, a'-1. \end{array}$$

où a et a' sont deux paramètres. On note $d = a + a'$.

Soit m un polynôme de $F[X]$ de degré inférieur à k qui vérifie $\deg(\mathrm{amp}(m)) \geq t$. Notons $M = \mathrm{amp}(m)$. Par définition de l'amplitude on a $m - R = 0 \mod M$. Par définition des g_i et des h_i on a donc :

$$\begin{array}{rcll} g_i(X, m(X)) & = & 0 \pmod{M^a} & \textit{pour } i = 0, \ldots, a-1, \\ h_i(X, m(X)) & = & 0 \pmod{M^a} & \textit{pour } i = 0, \ldots, a'-1. \end{array}$$

Remarque. Si le polynôme w est une combinaison linéaire de vecteurs de ces deux familles, il vérifie aussi l'égalité $w(X, m(X)) = 0 \pmod{M^a}$ pour tout m dans la solution du problème de décodage en liste.

On applique l'algorithme LLL polynomial au réseau engendré par les $g_i(X, X^k Y)$ et les $h_i(X, X^k Y)$. On note $w(X, X^k Y)$ le polynôme le plus court de la base réduite obtenue. Pour tout polynôme de degré plus petit que k on a $\deg_X w(X, m(X)) \leq \deg_X w(X, X^k Y)$, qui vérifie $\deg_X w(X, X^k Y) \leq |d(L)|/d$, où $d(L)$ est le déterminant du réseau considéré, car c'est le degré du vecteur de plus bas degré de la base LLL réduite. Si $\deg_X w(X, m(X)) < \deg_X M^a$ alors m est racine de w dans $F[X]$. Une condition suffisante est

$$\frac{|d(L)|}{d} < at.$$

Par définition des générateurs du réseau, la matrice du réseau est diagonale. On a donc facilement

$$|d(L)| = \frac{a(a+1)}{2} \cdot n + \frac{d(d-1)}{2} \cdot k.$$

La condition suffisante s'ecrit donc

$$\frac{a+1}{2d} \cdot n + \frac{d-1}{2a} \cdot k < t.$$

En fixant $d = a + a'$, on choisit a de façon à minimiser le terme de gauche de l'inégalité. On obtient le minimum pour a de l'ordre de $d \cdot \sqrt{\frac{k}{n}}$ et on obtient la condition suffisante $t > \sqrt{k \cdot n}$.

Dans [9], Sudan et Guruswami proposent également un algorithme pour décoder en liste les codes de Reed-Solomon. Les étapes sont identiques, mais la construction du polynôme bivarié se fait par la résolution d'un système linéaire. La borne trouvée sur les valeurs de t qui permettent à l'algorithme d'opérer est identique. Il serait intéressant de comparer les complexités et les temps d'exécution de ces deux algorithmes. Cependant, étant donné l'importance du décodage en liste, l'algorithme de Sudan et Guruswami a été optimisé de diverses manières. Il faudra en tenir compte au moment de la comparaison des temps d'exécution, ce qui pourra s'avérer délicat.

3.2.3 Implantation

Afin d'implanter cette méthode, on commence par implanter le calcul d'un polynôme à partir de ses racines, puis un algorithme qui réalise l'interpolation de Lagrange, puisqu'on a besoin de ces deux opérations pour calculer la matrice du réseau que l'on réduira grâce à une variante de l'algorithme LLL. On expliquera ensuite succinctement comment modifier l'algorithme de Paulus pour l'adapter à cette application. On s'est limité au calcul du polynôme bivarié w.

Idée de l'algorithme d'interpolation Dans ce qui suit ou note $M(n)$ le coût de la multiplication de deux polynômes de longueur n.

En fait nous utilisons un algorithme d'interpolation rapide, inspiré de l'algorithme présenté dans [17]. Etant donné un corps K, $u_0, \ldots, u_{n-1} \in K$ avec $i \neq j \Rightarrow u_i \neq u_j$ et $v_0, \ldots, v_{n-1} \in K$, il s'agit de calculer un polynôme $f \in K[X]$ de degré au plus $n-1$ tel que $\forall i, f(u_i) = v_i$. Le théorème d'interpolation de Lagrange nous assure de l'existence d'un tel polynôme. Il s'agit à présent de le calculer rapidement. Le théorème qui démontre l'existence du polynôme interpolé nous donne une formule explicite. Le polynôme que l'on cherche à calculer s'écrit

$$f = \sum_{i=0}^{n-1} v_i s_i \frac{m}{x - u_i},$$

où

$$m = \prod_{i=0}^{n-1} (x - u_i) \text{ et } s_i = \prod_{j \neq i} \frac{1}{u_i - u_j}.$$

On explique d'abord comment calculer les s_i rapidement. La dérivée de m peut s'écrire $m' = \sum_{j=0}^{n-1} m/(x - u_j)$, et comme $m/(x - u_i)$ s'annule en tous les points u_j avec $j \neq i$, on a

$$m'(u_i) = \left. \frac{m}{x - u_i} \right|_{x=u_i} = \frac{1}{s_i}.$$

Le polynôme m étant donné, le calcul de s_i équivaut à évaluer le polynôme m' en n points, puis à réaliser n inversions dans le corps de base.

Si l'on dispose d'une multiplication sous-quadratique, l'évaluation en n points d'un polynôme peut être réalisée récursivement, en un temps asymptotiquement plus rapide que celui obtenu en évaluant ce polynôme n fois en un point. L'idée de l'algorithme est de couper l'ensemble des points $\{u_0, \ldots, u_{n-1}\}$ en deux sous-ensembles et de procéder récursivement avec ces deux sous-ensembles. Cela conduit à la construction d'un arbre binaire de racine $\{u_0, \ldots, u_{n-1}\}$ et dont les feuilles sont les singletons $\{u_i\}$. On va construire un arbre correspondant à cette séparation de l'ensemble $\{u_0, \ldots, u_{n-1}\}$, de sorte que la racine de chaque sous-arbre soit le polynôme correspondant au produit de ses feuilles ; les feuilles de l'arbre sont les polynômes $X - u_i$. La construction de cette arbre nous fournit un calcul rapide d'un polynôme unitaire à partir de ses racines, et servira de données aux algorithmes d'évaluation en n points d'un polynôme et d'interpolation rapide.

Calcul d'un polynôme à partir de ces racines Une fois expliquée la manière de couper un ensemble de points en deux sous-ensembles, la construction de l'arbre binaire en découle facilement. Couper en deux sous-ensembles de tailles égales ou différant d'une unité semble le choix le plus naturel. Cependant dans [5], il est démontré que la manière optimale de procéder est de considérer un sous-ensemble dont la taille est la plus grande puissance de 2 strictement inférieure au cardinal de l'ensemble de départ, et son complémentaire. Dans le cas favorable où $n = 2^k$, les deux méthodes sont exactement équivalentes.

On peut dès lors donner l'algorithme qui construit l'arbre des produits des racines. On note $j = 2^{\lfloor \log n-1 \rfloor}$,$m_i = X - u_i$ pour $i < n$, $m_i = 1$ pour $n \leq i < 2^{j+1}$. On écrit $M_{k\ldots l} =$

Algorithme 6 Construction de l'arbre des sous-produits

Entrée: $\{u_0, \ldots, u_{n-1}\}$.
Sortie: L'arbre des sous produits des polynômes ayant pour racines $\{u_0, \ldots, u_{n-1}\}$.

```
if n = 1 then
  retourner X − u_0
else
  j ← 2^⌊log n−1⌋
  appeler l'algorithme récursivement pour calculer l'arbre des sous-produits de {u_0, ..., u_{j−1}}, A_g
  appeler l'algorithme récursivement pour calculer l'arbre des sous-produits de {u_j, ..., u_{n−1}}, A_d
  A ← (r(A_g) · r(A_d), A_g, A_d)
  retourner A
end if
```

$\prod_{i=k}^{l} m_i$. Enfin $A = (r, f_g, f_d)$ désigne l'arbre binaire de racine r, de fils gauche f_g et de fils droit f_d, $r(A)$ désigne la racine de A.

Étant donné que les propriétés sur les degrés des polynômes nous permettent de connaître *a priori* les largeurs des polynômes figurant dans l'arbre des sous-produits, on le représente par un tableau d'éléments et des calculs sur les indices du tableau nous permettent de savoir à quel polynôme tels coefficients correspondent. En mémoire l'arbre a donc la forme récursive suivante :

FIG. 7 – organisation en mémoire de l'arbre

Évaluation d'un polynôme en plusieurs points Une fois l'arbre des produits construit on peut aisément en déduire un algorithme rapide d'évaluation d'un polynôme en les points $\{u_0 \ldots u_{n-1}\}$. En effet on remarque que pour (q_0, r_0) (resp (q_1, r_1)) quotient et reste de la division euclidienne de f par $M_{0\ldots 2^j-1}$ (resp $M_{2^j\ldots 2^{j+1}-1}$) on a :

$$f(u_i) = \begin{cases} q_0(u_i)M_{0\ldots 2^j-1}(u_i) + r_0(u_i) = r_0(u_i) & \text{pour} \quad 0 \leq i < 2^j - 1 \\ q_1(u_i)M_{2^j\ldots n}(u_i) + r_1(u_i) = r_1(u_i) & \text{pour} \quad 2^j \leq i < n \end{cases}$$

D'où l'algorithme :

Algorithme 7 Évaluation d'un polynôme en plusieurs points

Entrée: $f \in K[X]$ de degré plus petit que n, $u_0, \ldots, u_{n-1} \in K$, et l'arbre des sous-produits associés.
Sortie: $f(u_0), \ldots, f(u_{n-1}) \in K$
  if $n = 1$ **then**
    retourner f
  else
    $j \leftarrow 2^{\lfloor \log n - 1 \rfloor}$
    $r_0 \leftarrow f \mod M_{1,\ldots,j-1}$
    $r_1 \leftarrow f \mod M_{j,\ldots,n-1}$
    appeler l'algorithme récursivement pour calculer $r_0(u_0), \ldots, r_0(u_{j-1})$
    appeler l'algorithme récursivement pour calculer $r_1(u_j), \ldots, r_1(u_{n-1})$
    renvoyer $r_0(u_0), \ldots, r_0(u_{j-1}), r_1(u_j), \ldots, r_1(u_{n-1})$
  end if

Combinaison linéaire On présente ici la fonction récursive qui est au cœur de l'algorithme d'interpolation rapide.

Algorithme 8 Combinaison linéaire

Entrée: $u_0, \ldots, u_{n-1}, c_0, \ldots, c_{n-1} \in K$ et l'arbre des sous produits correspondant aux $\{u_i\}$.
Sortie: $\sum_{0 \leq i < n} c_i \frac{m}{X - u_i}$ où $m = (X - u_0, \ldots, X - u_{n-1})$.
  if $n = 1$ **then**
    renvoyer c_0
  else
    $j \leftarrow 2^{\lfloor \log n - 1 \rfloor}$
    appeler l'algorithme récursivement pour calculer $r_0 = \sum_{0 \leq i < j} c_i \frac{M_{0,\ldots,j-1}}{X - u_i}$
    appeler l'algorithme récursivement pour calculer $r_1 = \sum_{j \leq i < n} c_i \frac{M_{j,\ldots,n-1}}{X - u_i}$
    renvoyer $M_{j,\ldots,n-1} r_0 + M_{0,\ldots,j-1} r_1$
  end if

Interpolation rapide Il ne reste plus qu'à agencer les algorithmes précédent afin d'obtenir l'algorithme d'interpolation rapide suivant :

Algorithme 9 Interpolation rapide

Entrée: $u_0, \ldots, u_{n-1} \in K$ tel que pour $i \neq j$ on ait $u_i \neq u_j$, et $v_0, \ldots, v_{n-1} \in K$.
Sortie: l'unique polynôme $f \in K[X]$ de degré plus petit que n tel que $\forall 0 \leq i < n, f(u_i) = v_i$.

appeler l'algorithme 3.2.3 avec pour entrée $u_0, \ldots, u_{n-1}$ pour calculer l'arbre des sous-produits M des u_i.
$m \leftarrow r(M)$
appeler l'algorithme 3.2.3 avec pour entrée $u_0, \ldots, u_{n-1}$ et M pour évaluer m' aux points $u_0, \ldots, u_{n-1}$.
for $0 \leq i < n$ **do**
  $s_i \leftarrow \frac{1}{m'(u_i)}$
end for
appeler l'algorithme 3.2.3 avec pour entrée $u_0, \ldots, u_{n-1}, v_0 s_0, \ldots, v_{n-1} s_{n-1}$ et M et **renvoyer** son résultat.

Modification apportée à l'algorithme LLL Dans le réseau qu'on est amené à réduire, tous les polynômes se situant au niveau de la i-ème coordonnée sont des multiples de $X^{k(i-1)}$. Au lieu de multiplier effectivement les coordonnées par ces monômes, on tient compte de ce facteur au moment du calcul de la norme des vecteurs de polynômes. On réduit ainsi le degré des polynômes apparaissant dans le réseau à réduire.

3.3 Factorisation de polynômes bivariés

On présente d'abord la manière d'appliquer l'algorithme LLL entier au cas de la factorisation de polynômes à coefficients entiers. Historiquement l'algorithme LLL a été introduit afin de résoudre ce problème en temps polynomial. Cependant en pratique il était moins efficace que l'algorithme de Cantor-Zassenhaus, algorithme exponentiel dans le pire cas mais qui se comporte bien en moyenne.

3.3.1 Méthode de Cantor-Zassenhaus

Le but est de factoriser un polynôme f de $\mathbb{Z}[X]$. On commence par le factoriser dans $\mathbb{F}_p[X]$, où p est un nombre premier bien choisi (notamment, il ne divise pas le discriminant de f). Pour ce faire on peut utiliser par exemple l'algorithme de Berlekamp. On relève ensuite cette factorisation dans $\mathbb{Z}/p^e\mathbb{Z}$ au moyen d'une méthode type Hensel, où e est choisi de sorte que tout facteur de f dans $\mathbb{Z}[X]$ ait ses coefficients plus petits que p^e. On connait des bornes qui garantissent l'existence d'un tel e. À cette étape, on a obtenu $f = \Pi_{i=1}^{s} f_i$ avec $f_i \in \mathbb{Z}/p^e\mathbb{Z}[X]$. Si l'on compare cette écriture à la factorisation de f dans $\mathbb{Z}[X]$, $f = \Pi_{i=1}^{l} g_i$, on voit que les g_i sont produits de certains f_j. On teste alors toutes les recombinaisons possibles, et on conserve celles qui divisent f dans $\mathbb{Z}[X]$. Cette dernière étape est exponentielle en s dans le pire des cas. On peut en général choisir p de telle sorte que s soit petit, mais il existe aussi des exemples de polynômes tel que pour tout bon premier p le nombre de facteurs obtenus au début de la dernière étape est linéaire en le degré de f, et donc où l'étape de recombinaison est effectivement exponentielle en le degré de f.

3.3.2 Méthode de van Hoeij

La méthode de van Hoeij améliore la phase de recombinaison en utilisant une méthode basée sur l'algorithme LLL. La remarque importante que fait van Hoeij est que le réseau L_f engendré par les vecteurs $(v_1, \ldots, v_k) \in \{0,1\}^s$ tel que $\Pi_{i=1}^s f_i^{v_{j,i}}$ soient les facteurs de f dans $\mathbb{Z}[X]$ est égal à l'ensemble des vecteurs w de $\mathbb{Z}^s$ tels que $\Pi_{i=1}^s f_i^{w_i}$ soit dans $\mathbb{Q}(X)$. L'algorithme de van Hoeij consiste à construire itérativement des réseaux L_i en partant de $L_0 = \mathbb{Z}^s$ et en les faisant décroître à chaque étape. C'est dans cette construction que va intervenir l'algorithme LLL. On utilise une forme linéaire petite sur L_f, les sommes de Newton. À chaque étape on applique l'algorithme LLL entier à un réseau construit à prtir de L_n, les vecteurs courts de la base obtenue engendrent un sous-réseau L_{n+1} contenant les vecteurs correspondant aux facteurs de f dans $\mathbb{Z}[X]$. On poursuit l'itération à partir de L_{n+1} jusqu'à ce que $L_N = L_f$.

On va tenter de reprendre les idées de cet algorithme de recombinaison dans le cas de polynomes bivariés en appliquant cette fois ci l'algorithme LLL de Paulus.

3.3.3 Cas bivarié

Commençons par poser le problème.

Dans ce qui suit K est un corps fini. Soit f un polynôme de $K[X,Y]$, de degré N en Y.

$$f = \sum_{i=0}^{N} c_i Y^i,$$

où $c_i \in K[X]$. On suppose que f est sans facteur carré. Ainsi f et f' sont premiers entre eux. On supposera que f est unitaire : $c_N = 1$. Notons, pour P polynôme irréductible de $K[X]$, $K[X]_{(P)}$ le completé $K[X]$ par (P). Soit un polynôme irréductible P tel que $f \mod P \in K[X]/(P)$ soit sans facteur carré. On note $d = \deg P$. On peut alors factoriser f dans $K[X]_{(P)}$ en factorisant f dans $K[X]/(P)$ par l'algorithme de Berlekamp, puis en appliquant un relèvement à la Hensel. On obtient donc

$$f = f_1 f_2 \cdots f_n.$$

Ici f_i est un polynôme irréductible unitaire de $K[X]_{(P)}[Y]$. Pour distinguer les facteurs dans $K[X]_{(P)}[Y]$ et les facteurs dans $K[X][Y]$, les f_i seront appelés *facteurs P-adiques* de f, et les facteurs de f dans $K(X)[Y]$ seront appelés *facteurs rationnels*. Par le lemme de Gauss, un facteur rationnel unitaire est dans $K[X][Y]$. Bien entendu on ne peut calculer sur un ordinateur que des approximations $\mathcal{C}^a(f_i) \in K[X][Y]$ des facteurs P-adiques avec une certaine précision a. Ces approximations sont appelés *facteurs modulaires*. Ils sont proches des f_i au sens de la norme P-adique.

Définition 11 *Soit $c \in K[X]_{(P)}$ et $0 \leq b \leq a$ des entiers. Le* reste $\mathcal{C}^a(c)$ *de c modulo P^a est l'unique polynôme de $K[X]$ congru à c modulo P^a.*

On définit encore $\mathcal{C}_b^a(c)$ comme $\mathcal{C}^{a-b}((c - \mathcal{C}^b(c))/P^b)$. Les définitions s'étendent au polynôme en appliquant $\mathcal{C}^a$ et $\mathcal{C}_b^a$ à chaque coefficient. On appelle encore $\mathcal{C}^a(c)$ une approximation de c avec précision a.

Si on voit c comme une série de produit d'éléments de $K[X]/(P)$ par des puissances de P alors $\mathcal{C}^a(c)$ est le polynôme qu'il reste après avoir retiré les puissances de P supérieures ou égales à a, $\mathcal{C}_b^a(c)$ est obtenu en supprimant les puissances de P supérieures ou égales à a et strictement inférieures à b, puis en divisant le polynôme restant par P^b.

Pour chaque facteur polynomial unitaire $g \in K[X,Y]$ de f il existe un sous-ensemble des facteurs P-adiques $S \subseteq \{f_1, \ldots, f_n\}$ tel que

$$g = \prod_{f_i \in S} f_i.$$

Réciproquement, si S est un sous-ensemble de $\{f_1, \ldots, f_n\}$ alors $g = \prod_{f_i \in S} f_i$ est un facteur rationnel de f si et seulement si $g \in K[X][Y]$, c'est à dire si les coefficients de g (qui a priori appartiennent à $K[X]_{(P)}$) sont dans $K[X]$.

On a donc le problème combinatoire suivant : comment trouver les sous-ensembles S de $\{f_1, \ldots, f_n\}$ pour lesquels le produit des éléments de S sont à coefficients polynomiaux. L'algorithme de Berlekamp-Zassenhaus essaie tous les sous-ensembles possibles, si bien que sa complexité est proportionnelle à 2^n.

Ce qui soulève immédiatement la question suivante : étant donné un tel sous-ensemble S, comment un ordinateur peut-il décider si le produit g a des coefficients polynomiaux, étant donné que l'on ne peut calculer que des approximation des f_i avec une précision finie a, ce qui suffit pour trouver $\mathcal{C}^a(g)$, mais ne suffit pas toujours à déterminer g, si g est à coefficients P-adiques. Ce problème est résolu de la manière suivante. On calcule une borne B telle que l'on peut prouver que le degré des coefficients d'un facteur polynomial de f est toujours plus petit que B. On choisit alors a tel que $ad > B$. Les trois propositions suivantes sont alors équivalentes :

1. $g \in K[X][Y]$;
2. $g = \mathcal{C}^a(g)$;
3. $\mathcal{C}^a(g)$ divise f dans $K[X][Y]$.

Pour chacun des 2^n sous-ensembles de $\{f_1, \ldots, f_n\}$, ou 2^{n-1} si on n'effectue pas les calculs sur les complémentaires des ensembles déjà essayés, on doit tester si $\mathcal{C}^a(g)$ divise f dans $K[X][Y]$. Ceci peut être fait rapidement, mais comme le nombre d'ensembles à tester est exponentiellement grand, l'algorithme de Berlekamp-Zassenhaus est a priori exponentiel. En pratique pourtant il se comporte bien, car il n'est pas exponentiel en le degré du polynôme N, mais en le nombre de facteur P-adiques n, qui habituellement est bien plus petit que N. Mais il existe des polynômes pour lesquels n est effectivement grand, et dans ce cas l'algorithme est exponentiel.

Le premier algorithme en temps polynomial a été proposé par A. K. Lenstra dans [11]. Au lieu de considérer tous les sous-ensembles S, il choisit un facteur P-adique, disons f_1, puis déterminent (s'il existe, c'est à dire si f est réductible) un polynôme $g \in K[X][Y]$ de degré $< N$ tel que f_1 divise g. Alors pgcd(f,g) est un facteur non trivial de f. De cette manière on évite le problème combinatoire exposé ci-dessus. Pour ce faire on met en oeuvre l'algorithme LLL polynomial.

Cependant pour construire les facteurs polynomiaux, on n'opèrent plus en multipliant des facteurs modulaires mais on utilise une autre méthode, moins rapide. Ceci explique qu'en

général l'algorithme de Berlekamp-Zassenhaus se comporte mieux que l'algorithme de A. K. Lenstra pour factoriser des polynômes bivariés sur un corps fini.

Dans ce qui suit on présente la méthode que van Hoeij à mis en oeuvre pour résoudre le problème combinatoire de la recombinaison. Il utilise un algorithme LLL, ici non pas pour éviter le problème combinatoire, mais pour le traiter. L'approche a deux avantages. Un sous-ensemble S peut être représenté par un vecteur 0-1. Un vecteur des coefficients de g peut avoir une taille beaucoup plus importante. De plus la taille du réseau sera proportionnelle à n et non plus à N comme dans [11].

3.3.4 Fonction Trace

Définition 12 *La i-ème trace $Tr_i(g)$ d'un polynôme g est définie comme la somme des puissances i-ème des racines g, comptées avec leur multiplicité.*

On l'appelle aussi i-ème somme de Newton. Il est clair que

$$Tr_i(f_1) + Tr_i(f_2) = Tr_i(f_1 f_2)$$

pour tous polynômes f_1, f_2. Considérons le polynôme

$$g = (Y - x_1)(Y - x_2) \cdots (Y - x_m)$$

On peut encore écrire

$$g = Y^m + \tilde{E}_1 Y^{m-1} + \tilde{E}_2 Y^{m-2} + \cdots + \tilde{E}_m$$

où $\tilde{E}_i = (-1)^i E_i$ et $E_i = E_i(x_1, \ldots, x_m)$ est le i-ème polynôme symétrique élémentaire en les variables $x_1, \ldots, x_m$. Le i-ème polynôme des puissances $P_i(x_1, \ldots, x_m)$ est défini comme $x_1^i + x_2^i + \cdots + x_m^i$. On remarque que $Tr_i(g) = P_i$. On a le résultat classique suivant :

$$K[E_1, \ldots, E_m] = K[P_1, \ldots, P_m],$$

en d'autres termes les E_i peuvent être exprimés comme des polynômes en P_i et vice versa.

Lemme 6 *Si un polynôme unitaire g de degré m est à coefficients dans $K(X)$ alors $Tr_i(g) \in K(X)$ pour tout $i \in \{1, \ldots, m\}$. La réciproque est vraie en caractéristique 0, ou encore dans le cas où le degré de g est strictement plus petit que la caractéristique du corps.*

Ceci découle simplement des identités de Newton

$$P_i = -i\tilde{E}_i - \sum_{k=1}^{i-1} P_k \tilde{E}_{i-k},$$

et

$$i\tilde{E}_i = -P_i - \sum_{k=1}^{i-1} P_k \tilde{E}_{i-k}.$$

La condition sur le degré de g vise à pouvoir inverser i dans la deuxième formule. On se servira de la première relation pour calculer rapidement $Tr_i(g)$.

3.3.5 Facteurs polynomiaux : conditions nécessaires

On se place dans le cas où $N < p$, p étant la caractéristique du corps K.
Posons maintenant pour M entier ≥ 1 $Tr_{1...M} = (Tr_1, \ldots, Tr_M)^T$.

$$Tr_{1...d}(g) = \begin{pmatrix} Tr_1(g) \\ Tr_2(g) \\ \vdots \\ Tr_M(g) \end{pmatrix}.$$

On va commencer par donner un lemme qui fournit une condition nécessaire et suffisante pour qu'un produit g soit polynomial.

Lemme 7 *Soit f un polynôme unitaire de degré N dans $K[X][Y]$. On note $m = \lfloor N/2 \rfloor$. Soit F un corps. Si $m < p$, p étant la caractéristique de K, pour tout facteur unitaire $g \in F[X]$ de f on a l'équivalence entre les propositions suivantes.*

1. $g \in K[X][Y]$;
2. $Tr_{1...m}(g) \in K[X]^m$;
3. $Tr_{1...m}(g) \in K(X)^m$.

La condition imposée sur le degré de f assure que la réciproque du lemme 6 est vraie. On peut sûrement s'affranchir de cette limitation en utilisant un résultat de théorie des invariants nous donnant une égalité entre le corps des fractions engendré par les premièrs polynômes des puissances et le corps des fractions engendré par les premiers polynômes symétriques élémentaires en caractéristique non nulle. Dans ce cas il faudra certainement utilisé un plus grand nombre de polynômes des puissances. Il serait tout à fait intéressant de pousser la réflexion dans ce sens. On se borne par la suite à notre cas particuliers. On pourra facilement étendre la suite du raisonnement au cas où un tel résultat est disponible.

On va maintenant affaiblir le lemme précédent afin de n'obtenir qu'une condition suffisante. Soit s et d deux entiers positifs, et A une matrice $s \times d$ à valeurs polynomiales. On note

$$T_A = ATr_{1...d} \quad , \quad T_A(g) = A \begin{pmatrix} Tr_1(g) \\ Tr_2(g) \\ \vdots \\ Tr_d(g) \end{pmatrix}.$$

Grâce à la matrice A on utilise un vecteur avec s coefficients, avec s petit, au lieu d'un vecteur à d coefficients, d pouvant être très grand. De plus cela permet de condenser des informations sur un grand nombre de $Tr_i(g)$ dans un petit vecteur.

Lemme 8 *Si $g \in K[X]_{(P)}$ est un facteur unitaire de f alors*

$$g \in K[X][Y] \Rightarrow T_A(g) \in K[X]^s$$

Si la condition suivante est vérifiée : «l'espace vectoriel engendré par les lignes de A contient les $\lfloor N/2 \rfloor$ éléments $\begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix}$, $\begin{pmatrix} 0 & 1 & \cdots & 0 \end{pmatrix}$, ...» alors la réciproque est vraie.

Penchons nous à présent sur le problème des approximations. Soit S un sous-ensemble des facteurs P-adiques et g le produit des éléments de S. Alors

$$T_A(g) = \sum_{f_i \in S} T_A(f_i).$$

Ainsi, une condition nécessaire pour que g soit un facteur polynomial est que la somme des $T_A(f_i)$, $f_i \in S$ soit un vecteur de polynômes de $K[X]$. Cependant on ne peut déterminer $T_A(f_i)$ qu'approximativement, à la précision a. En fait si B est une borne sur les degrés des coefficients des facteurs polynomiaux de f, alors les identités de Newton fournissent des bornes sur les degrés des $Tr_i(g)$ pour g facteur polynomial de f. On en déduit des bornes sur les coefficients de $T_A(g)$. Étant donné S, on peut calculer $\mathcal{C}^a(T_A(g))$ et une condition nécessaire pour que $g \in K[X][Y]$ est que l'approximation à la précision a satisfasse la limite dans chaque ligne.

La seule chose qui est importante est donc de savoir si $\mathcal{C}^a(T_A(g))$ satisfait la borne. Pour ce faire il n'est pas nécessaire de calculer la valeur précise de $T_A(g)$.

On commence par calculer une borne B_i pour chaque ligne i de $T_A(g)$, c'est à dire que le degré de la i-ème coordonnées de $T_A(g)$ doit être $< B_i$ pour tout facteur polynomial g de f. Puis on choisit des entiers $b = (b_1 \cdots b_s)$ tels que $B_i < \deg P^{b_i}$.

Définition 13 *Pour g, un facteur P-adique unitaire de f, on définit $T_A^b(g) \in K[X]_{(P)}$ de la manière suivante : soit r la i-ème coordonnée de $T_A(g)$. Soit $\bar{r}$ l'approximation de r à la précision b_i. Alors $r - \bar{r}$ est divisible par P^{b_i}, $u = (r - \bar{r})/P^{b_i}$ est dans $K[X]_{(P)}$. La i-ème coordonnée de $T_A^b(g)$ est par définition u.*

Si g est un facteur polynomial de f alors le degré de la i-ème coordonnée de $T_A(g)$ est borné par B_i et est donc plus petite que le degré de P^{b_i}. Donc la i-ème coordonnées de $T_A(g)$ est égale à son approximation à la précision b_i. Donc $T_A^b(g)$ est nul, ce qui prouve la première partie du lemme suivant.

Lemme 9 *Soit $g \in K[X]_{(P)}$ un facteur unitaire de f. Alors*

$$g \in K[X][Y] \Rightarrow T_A^b(g) = 0.$$

De plus, si A satisfait le condition du lemme 8, alors la réciproque est vraie.

On remarque que si f_j a été calculé à la précision a, on peut calculer la i-ème coordonnée de $T_A^b(f_j)$ à la précision $a - b_i$. Il faut donc que a soit supérieur que b_i, en particulier plus grand que B_i/d. Si ce n'est pas le cas, il faut réappliquer la méthode de Hensel pour augmenter la valeur de a.

Contrairement au cas de la caractéristique 0 dans lequel van Hoeij s'était placé pour factoriser des polynômes de $\mathbb{Z}[X]$, T_A^b est encore additive. De là on tire une version plus appropriée du lemme 9.

Lemme 10 *Soit S un sous-ensemble de $\{f_1, \ldots, f_n\}$ et g le produit des éléments de S. Si $g \in K[X][Y]$ alors*

$$\sum_{f_i \in S} T_A^b(f_i) = 0.$$

De plus si A satisfait la condition du lemme 8 alors la réciproque est vraie.

Choisissons à présent des entiers a_i tels que $b_i < a_i$. Soit $\bar{c}_{j,i}$ la i-ème coordonnée de $T_A(f_j)$ et soit $\tilde{c}_{j,i}$ la i-ème coordonnée de $T_A^b(f_j)$. Posons à présent

$$c_{j,i} = \mathcal{C}_{b_i}^{a_i}(\bar{c}_{j,i}) = \mathcal{C}^{a_i - b_i}(\tilde{c}_{j,i})$$

et définissons $\mathcal{C}_j \in K[X]^s$ comme

$$\mathcal{C}_j = (c_{j,1} \cdots c_{j,s})^T.$$

La i-ème entrée de $\mathcal{C}_j$ est une approximation à la précision $a_i - b_i$ de la i-ème coordonnée de $T_A(f_j)$. On peut la calculer rapidement à partir de $\mathcal{C}^a(f_j)$ dans le cas où $a \geq a_i$.

Théorème 4 *Soit f un polynôme unitaire sans facteur carré de $K[X][Y]$ et $f_1, \ldots, f_n$ les facteurs P-adiques irréductibles. Pour tout $S \subseteq \{f_1, \ldots, f_n\}$, si le produit g des éléments de S est un facteur polynomial de f alors*

$$\sum_{i=1}^{n} v_i \mathcal{C}_i = 0, \tag{1}$$

où

$$v_i = \begin{cases} 1 & si\ f_i \in S \\ 0 & sinon \end{cases}$$

Si A satisfait la condition du lemme 8 alors la réciproque du théorème est vraie.

Ceci découle du lemme 10, dont ce théorème est une réécriture.

3.3.6 Le réseau "sac à dos" et l'algorithme de recombinaison de van Hoeij

Soit $g_1, \ldots, g_r$ les facteurs irréductibles unitaires de f dans $K[X][Y]$. On note $w_1, \ldots, w_r$ les vecteurs de $\{0,1\}^n$, où w_k est défini par $w_k = (v_1, \ldots, v_n)$ tels que $g_k = \prod f_i^{v_i}$. On peut considérer ces vecteurs comme des vecteurs de $K[X]^n$. On note W le réseau de $K[X]^n$ engendré par $\{w_1, \ldots, w_k\}$. L'ensemble des w_i est une base du réseau. Cette base est sous forme «échelon réduite». Déterminer la base échelon réduite de W est équivalent à résoudre le problème combinatoire rencontré dans l'algorithme de Berlekamp-Zassenhaus.

On va utiliser les notations suivantes. $L \subseteq K[X]^n$ est un réseau, B_L une base de L. La matrice dont les colonnes sont les éléments de B_L est notée (B_L), et la forme échelon réduite de cette matrice est notée $\mathrm{rref}(B_L)$. Si on dispose d'une base B_W de W le problème combinatoire est résolu car $\{w_1, \ldots, w_r\}$ sont les colonnes de $\mathrm{rref}(B_W)$.

Lemme 11 *Soit L un réseau tel que*

$$W \subseteq L \subseteq K[X]^n. \tag{2}$$

Soit R =rref(B_L). Alors $L = W$ si et seulement si les deux conditions suivantes sont vérifiées :

1. *Chaque ligne de R ne contient qu'un seul coefficient non nul égal au polynome constant 1*

2. Si $(v_1 \cdots v_n)^T$ *est une colonne de* R *alors* $g = \prod f_i^{v_i} \in K[X][Y]$.

Si on a un réseau L qui satisfait l'équation (2) alors on peut tester si $L = W$ en testant si les conditions 1. et 2. sont vérifiées. La condition 2. peut être vérifiée comme dans l'algorithme de Berlekamp-Zassenhaus, en calculant le reste du produit modulo P^a et en vérifiant que le résultat divise f dans $K[X][Y]$.

Au départ, on prend $L = K[X]^n$, ainsi on est certain que l'équation (2) est vérifiée au départ. Si on s'assure que cette équation reste vraie au cours de l'algorithme, alors à chaque étape on peut tester si $L = W$ grâce au lemme 11. Supposons $L \neq W$. Le principe de l'algorithme est de calculer un nouveau réseau L' tel que

$$W \subseteq L' \subseteq L,$$

de telle sorte que L' vérifie également l'équation (2). On remplace alors L par L', et on répète cette étape tant que les conditions 1. et 2. ne sont pas vérifiées. Une fois qu'elles sont vérifiées, la vérification de la condition 2. livre tous les facteurs polynomiaux irréductibles de f.

1. **Cet algorithme termine-t-il ?** Pour prouver la terminaison, il faut prouver que les conditions 1. et 2. finissent par être réalisées.
2. **Retourne-t-il le bon résultat ?** Si l'algorithme termine, comme l'équation (2) reste vraie au cours de l'algorithme, on obtient bien tous les facteurs polynomiaux irréductibles de f.

Choisissons une matrice A de taille $s \times m$ et des entiers a_i et b_i, avec comme on l'a vu $a \geq a_i > b_i > B_i/d$. On va montrer comment l'algorithme de van Hoeij calcule un nouveau réseau $L' \subseteq L$, qu'on espère de plus petite dimension, et qui contient les solutions $(v_1 \cdots v_n)$ de l'équation (1). L'algorithme sera correct, car $w_1, \ldots, w_n$ vérifient l'équation (1), et donc $W \subseteq L'$.

Si $\dim(L') = \dim(L)$ alors on utilisera une autre matrice A. Van Hoeij montre dans [16] qu'au bout du compte on obtiendra un réseau L' de dimension strictement plus petite. On remplace alors L par L', on vérifie les conditions 1. et 2. pour tester si $L = W$, et au bout d'un nombre fini de telles étapes l'algorithme termine.

On remarque que le théorème 1 affirme que W est inclus dans le noyau de T_A^b pour tout A et b convenable. À chaque étape on cherche donc l'intersection de L et du noyau de T_A^b et si sa dimension est plus petite que celle de L alors on a trouvé un sous réseau L' convenable.

Ce calcul du noyau d'une application linéaire sur un $K[X]$-module peut être réalisé grâce à l'algorithme LLL polynomial. En effet quand on réduit une base du réseau engendré par les vecteurs de la matrice décrivant l'application linéaire, si son noyau n'est pas vide des vecteurs s'annuleront au cours de la réduction. On peut de plus obtenir la matrice de passage entre la base initiale et la base réduite à peu de frais au cours de la réduction, en réalisant les opérations effectuées sur une matrice auxiliaire, initialement égale à l'identité. Les colonnes correspondant aux colonnes qui s'annulent au cours de l'algorithme donnent les vecteurs de base du noyau de l'application linéaire.

3.3.7 Conclusion

Nous nous sommes limités à une étude théorique de cet algorithme. Ceci nous a permis de constater les deux principales différences entre la version de l'algorithme de van Hoeij appliquée à la factorisation de polynômes de $\mathbb{Z}[X]$ et celle que l'on désire appliquer au polynômes de $K[X][Y]$ avec K un corps fini. Ces deux différences proviennent du fait que dans un cas on travaille en caractéristique nulle, et dans l'autre cas en caractéristique positive.

Dans le cas de la caractéristique nulle on a une correspondance facile entre polynômes symétriques élémentaires et polynômes des puissances, donnée par les formules de Newton. En caractéristique positive, les formules de Newton ne donnent plus une telle équivalence et on a besoin de faire appel à des résultats plus poussés de la théorie des invariants pour ne pas avoir à imposer des limitations sévères au degré des polynômes que l'on peut factoriser.

Dans le cas de la caractéristique positive, la fonction T_A^b reste additive. Ceci simplifie considérablement la phase de recherche du réseau, puisqu'elle est ramenée à des calculs de noyau de fonctions linéaires.

Perspectives. Il serait intéressant de réaliser une implantation de cet algorithme. La principale difficulté est que cet algorithme résout un problème soulevé par un autre algorithme. Un implantation du relèvement à la Hensel peut s'avérer nécessaire.

Il sera intéressant de comparer les résultats de cet algorithme avec celui de Berlekamp-Zassenhaus et avec celui de A. K. Lenstra.

4 Synthèse

On a présenté l'algorithme LLL de Paulus, qui permet d'obtenir à partir d'une base d'un réseau polynomial une base réduite. On a rapproché les notions introduites par Paulus de la notion de matrice sous forme de Popov faible introduite par T. Mulders et A. Storjohann. La complexité de l'algorithme de Paulus est $O(nmrd^2)$, n étant la dimension des vecteurs du réseau, m le nombre de vecteur dans le systèeme générateur initial, r le rang de la matrice et d une borne sur les degrés des polynômes intervenant dans la matrice du réseau initial. On a implanté une version de cet algorithme en C. Cette version est modulaire, dans les sens où la représentation des coefficients de corps est séparée du reste du code, ce qui permet d'en changer facilement. On a donné des valeurs de temps d'exécution en faisant varier les divers paramètres, confirmant ainsi les bornes théoriques, et donnant un ordre d'idée de l'efficacité de l'implantation.

On a présenté ensuite trois applications à l'algorithme LLL polynomial.

Dans un premier temps, on a implanté la version modifiée de l'algorithme LLL utilisée par A. Basiri et J-C. Faugère pour obtenir des bases de Gröbner d'idéaux de polynômes bivariés pour un ordre monomial à partir d'une base de Gröbner pour un autre ordre monomial. Notre implantation gagne un facteur 100 sur l'implantation MAPLE effectuée à des fins de démonstration pour l'article, sur les exemples traités. Cependant on constate que le déroulement de l'algorithme conduit à une divergence des normes des vecteurs au cours de l'exécution, ce qui peut s'avérer problématique en ce qui concerne la mémoire nécessaire à l'exécution. On a ensuite tenter d'accélérer l'exécution en implantant des primitives d'arithmétique rapide

sur les polynômes, mais notre travail s'est révélé infructueux, car les arguments passés à ces fonctions au cours du calcul ont des tailles très déséquilibrées.

On s'est ensuite intéressé à l'adaptation d'applications de LLL entier au cadre polynomial. Nous avons ainsi donné un algorithme de décodage en liste des codes de Reed-Solomon utilisant un avatar de LLL polynomial, que nous avons en partie implanté. Puis sur un plan plus théorique on a repris les idées que van Hoeij met en oeuvre pour obtenir un algorithme de factorisation des polynômes de $\mathbb{Z}[X]$, afin d'obtenir un algorithme analogue pour la factorisation des polynômes de $K[X][Y]$, K corps fini.

Il pourrait être intéressant de poursuivre le travail entamée sur l'adaptation de l'algorithme de van Hoeij au cas de polynômes bivariés sur un corps fini. Il y a en fait deux directions à cette poursuite :

- une direction mathématique. Il s'agit de d'obtenir, où de retrouver dans la littérature, un résultat de théorie des invariants qui nous permettrait de nous affranchir d'une limitation que nous avons imposée aux polynômes que l'on peut traiter par l'algorithme que nous avons exposé ;
- une direction technique. Il s'agit d'implanter cet algorithme, et de comparer son efficacité avec des algorithmes de factorisations existants.

D'autre part, en ce qui concerne le problème du décodage en liste des codes de Reed-Solomon, la comparaison entre l'algorithme de Sudan et Guruswami et l'algorithme que nous avons implanté devrait être menée à son terme.

Enfin, en ce qui concerne le changement d'ordre monomial dans les bases de Gröbner, on pourrait étudier l'ordonnancement des opérations effectuées au cours de l'algorithme afin d'essayer d'éviter une explosion des normes des vecteurs de la base en cours d'exécution. On pourra de plus essayer de l'accélérer en branchant une division rapide se comportant bien lorsque ses arguments ont des tailles déséquilibrées.

Références

[1] Abdolali Basiri and Jean-Charles Faugère. Changing the ordering of Gröbner Bases with LLL : case of Two Variables. Proc ISSAC 2003, à paraître.

[2] C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. PARI-GP. http ://www.parigp-home.de/.

[3] Dan Boneh. Finding smooth integer in short intervals using CRT decoding. *Journal of Computer and System Sciences (JCSS)*, 64 :768–784, 2002.

[4] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal* . PhD thesis, Innsbruck, 1965.

[5] A. Enge, P. Gaudry, G. Hanrot, and P. Zimmermann. Reconstructing a Polynomial from its Roots. Travail en cours.

[6] J-C. Faugère, P. Gianni, D. Lazard, and T. Mora. Efficient Computation of zero-dimensionnal Gröbner Bases by change of ordering. *Journal of Symbolic Computation*, 16 :329–344, 1993.

[7] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner Bases (f4) . *Journal of Pure and Applied Algebra*, 139(1-3) :61–88, June 1999.

[8] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner Bases without Reduction to Zero (f5) . In ACM Press, editor, *Proceedings of ISSAC*, pages 75–83, July 2002.

[9] V. Guruswami and M.Sudan. Improved Decoding of Reed-Solomon and Algebraic-Geometry Codes, May 1999.

[10] A. K. Lenstra. Factoring multivariate polynomials over finite fields. *Journal of Computer and System Sciences (JCSS)*, 30(2), 1985.

[11] A. K. Lenstra. Factoring Multivariate Polynomials over Finite Fields. *Journal of Computer and System Sciences (JCSS)*, 30(2) :235–248, 1985.

[12] A. K. Lenstra, H. W. Lenstra Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261, 1982.

[13] T. Mulders and A. Storjohann. On lattice reduction for polynomial matrices. *Journal of Symbolic Computation 35*, 2003. 377-401.

[14] Sachar Paulus. Lattice Basis Reduction in Function Fields. In *Lecture Notes in Computer Science 1423 (Algorithmic Number Theory)*, March 1998.

[15] V. Popov. Some Properties of control systems with irreducible matrix transfer functions. *Lecture Notes in Mathematics*, 144, 1969.

[16] Mark van Hoeij. Factoring polynomials and the knapsack problem. *Journal of Number Theory*, 95 :167–189, 2002.

[17] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[18] Paul Zimmermann. Arithmétique en Précision Arbitraire, 2001. Rapport de Recherche INRIA RR-4272.